REPORT DOCUMENTATIC AD-A280 633

AD-A280 633

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE April 1994	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE Techniques for Real-Time Parallel Processing: Sensor Processing Case Studies	5. FUNDING NUMBERS F19628-94-C-0001
6. AUTHOR(S) Richard A. Games, John D. Ramsdell, Joseph J. Rushanan	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The MITRE Corporation 202 Burlington Road Bedford, MA 01730-1420	8. PERFORMING ORGANIZATION REPORT NUMBER MTR 93B0000186
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) ESC Hanscom Air Force Base Bedford, MA 01731	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The software development process for parallel processors is investigated with a focus on real-time sensor processing applications. Concepts that are relevant to real-time parallel processing are introduced including: a definition of scalable dataflow graphs motivated by the need to meet a fixed throughput constraint for varying problem sizes, an algorithm classification that makes explicit the impact that data dependencies have in real-time implementations, and a real-time implementation strategy that decomposes the most problematic algorithms into compositions of more predictable constituents and then uses scalable dataflow graphs and parallel processing to recover timing predictability by mapping data-dependent timing uncertainties into the spatial dimension (processors). Two case studies apply these ideas: an implementation of the Modified Gram-Schmidt (MGS) algorithm on a MasPar MP1 and an implementation of the joint probabilistic data association (JPDA) algorithm on a Thinking Machines CM-2. The JPDA case study includes a SISAL implementation to illustrate the advantages of functional programming for these applications.

14. SUBJECT TERMS Parallel processors, scalable dataflow graphs, algorithms	15. NUMBER OF PAGES 71
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlimited

Best
Available
Copy

Techniques for Real-Time Parallel Processing: Sensor Processing Case Studies

Richard A. Games
John D. Ramsdell
Joseph J. Rushanan

DTC QUALITY LNSPECTED

94-18923

MITRE

Bedford, Massachusetts

94 6 23 029

Techniques for Real-Time Parallel Processing: Sensor Processing Case Studies

Richard A. Games
John D. Ramsdell
Joseph J. Rushanan

MITRE

Bedford, Massachusetts
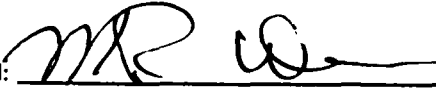
Department Approval: _____

M. R. Weiss
Acting Department Head, D051

MITRE Project Approval: _____

R. A. Games
Project Leader, 7150

ii

ABSTRACT

The software development process for parallel processors is investigated with a focus on real-time sensor processing applications. Concepts that are relevant to real-time parallel processing are introduced including: a definition of scalable dataflow graphs motivated by the need to meet a fixed throughput constraint for varying problem sizes, an algorithm classification that makes explicit the impact that data dependencies have in real-time implementations, and a real-time implementation strategy that decomposes the most problematic algorithms into compositions of more predictable constituents and then uses scalable dataflow graphs and parallel processing to recover timing predictability by mapping data-dependent timing uncertainties into the spatial dimension (processors). Two case studies apply these ideas: an implementation of the Modified Gram-Schmidt (MGS) algorithm on a MasPar MP1 and an implementation of the joint probabilistic data association (JPDA) algorithm on a Thinking Machines CM-2. The JPDA case study includes a SISAL implementation to illustrate the advantages of functional programming for these applications.

iii

Accession For

NTIS CRA&I

DTIC TAB

Unannounced

Justification

By

Distribution /

Availability

Dist Avail and / or Special

A-1

ACKNOWLEDGMENTS

- MasPar is a registered trademark of MasPar Computer Corporation.

- DECstation and ULTRIX are trademarks of Digital Equipment Corporation.

- Connection Machine, Thinking Machines, and C* are registered trademarks of Thinking Machines Corporation. CM-2 and CM-5 are trademarks of Thinking Machines Corporation.

- Sun, SunOS, Sun-4, SPARCstation, and Sparc10 are trademarks of Sun Microsystems, Inc.

- UNIX is a trademark of UNIX System Laboratories.

- The X Window System is a trademark of the Massachusetts Institute of Technology.

- Paragon is a trademark of the Intel Corporation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

SECTION 1

INTRODUCTION

The process of specifying, designing, manufacturing, and supporting complex digital systems, particularly real-time embedded signal and data processors, must change if future system requirements are to be met within shrinking military budgets. High performance computing can play a significant role in the military's evolving seamless design methodology to rapidly produce systems with reduced life-cycle costs. If the computational requirements of current and future real-time embedded systems can be satisfied by emerging programmable massively parallel computers, then costly application-specific processors and disparate data processors can be replaced by a single homogeneous, scalable, programmable computing platform that is designed to track the progression of commercial technology.

There have been significant advances in hardware technology that make such a high performance computing solution possible: processing, I/O, and memory capabilities are continuing to improve. The packaging problems associated with embedded applications are currently being addressed by a variety of ARPA research and development programs. In particular, the Militarized Touchstone program [1] is producing embedded high performance computers that incorporate commercial microprocessors running the same software as their commercial counterparts.

However, software technology for high performance computing has been lagging. There is a need for improved programming environments and tools that produce portable implementations with increased hardware utilization. The problem becomes more difficult for hard real-time implementations that must meet strict timing deadlines. This report investigates the software development process for parallel processors with a focus on real-time sensor processing applications.

APPLICATIONS: SENSOR PROCESSING

The problems with the current software development process must be made explicit by applying the current process to motivating applications and assessing the resulting performance as a function of level of software development effort. In this paper we focus on sensor processing since it is prevalent throughout the DOD and has dual military-commercial potential. The impact of high performance computing on sensor processing will grow in the future, affecting many current

1
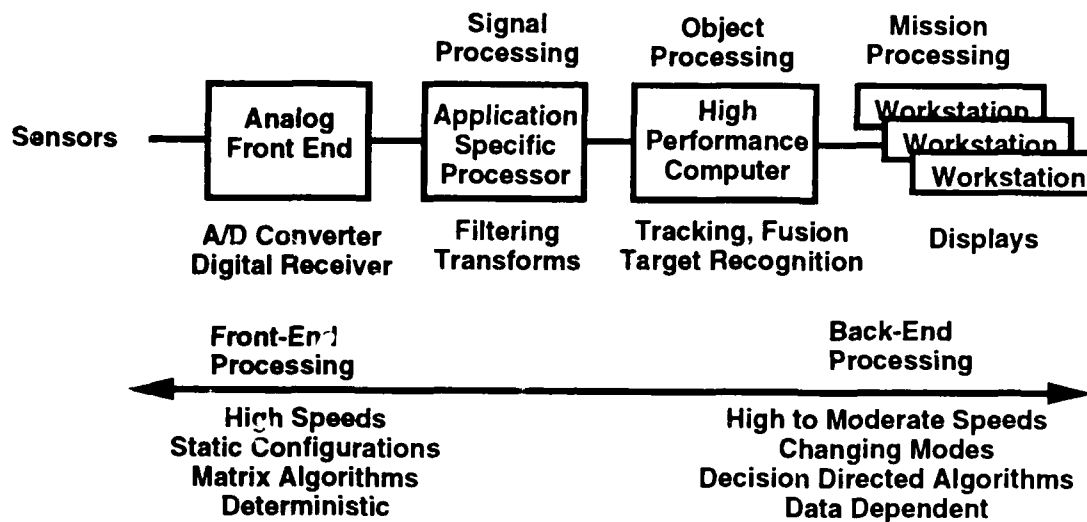
Figure 1. Sensor Processing Chain

and future military and commercial programs. Current airborne surveillance systems such as the Airborne Warning and Control System (AWACS) [2] and the Joint Surveillance and Target Attack Radar System (Joint STARS) [3] are prime candidates. Emerging sensor applications such as space-time adaptive processing [4], which involves computationally complex algorithms, and wide-area, high-resolution surveillance [5], which involves massive amounts of data, will require supercomputing performance to satisfy the application's real-time requirements.

Sensor processing involves a mix of processing types: signal processing, object processing, and mission processing (see Figure 1). An initial analog-to-digital conversion process converts the analog sensor inputs to digital form for subsequent digital processing. Signal processing is then applied to this digital data to accomplish a variety of functions such as the removal of interference, the formation of images, and the detection of signals of interest. Subsequent object processing involves functions such as automatic target recognition and tracking. The final mission processing phase involves presenting data to and assisting a decision maker and then implementing or communicating decisions that affect, for instance, the allocation of resources. The point of our work is to determine whether high performance computing platforms can meet the range of processing requirements.

2

Front-end signal processing is characterized by fixed dataflow patterns with high data rates. The algorithms usually involve blocks of data, for example, the fast Fourier transform or matrix factorization routines from linear algebra. In the past, technology limitations dictated that special purpose hardware was required for signal processing, often using pipelined systolic array architectures. The current challenge is to achieve the necessary processing efficiency with programmable general purpose hardware to meet the hardware size, weight, and power requirements of embedded applications.

In contrast, the object processing stage is characterized by reduced data rates, but with dataflow patterns and computational loads that often change depending on the input from the sensor. For example, tracking algorithms may require an amount of computation based on the number of targets. In the past, general purpose programmable computers have been used to implement object processing. Often in these implementations, a number of processing tasks competed for a single computational resource, and this made guaranteeing the real-time performance a difficult problem. With the advent of parallel processing, the processing resources have increased, but new problems involving mapping and scheduling the computation and communication tasks on multiple processing and (usually more limited) network resources continue to make guaranteeing predictable real-time performance difficult. Data dependencies in this processing phase imply that more challenging and costly dynamic allocation of resources are required to reduce the hardware size, further exacerbating the problem of meeting real-time requirements.

ORGANIZATION AND SUMMARY OF RESULTS

Section 2 describes the fundamental concepts used in the case studies that comprise Sections 3 and 4. Many of the concepts, such as *algorithms* and *dataflow graphs*, are well known and descriptions are included in Section 2 to establish terminology and to make the document more self-contained. However, Section 2 also refines old concepts and introduces new ones that are particularly relevant for real-time parallel processing. These include:

- A discussion of how the parallelization strategies of splitting, replication, and pipelining are applied to meet latency and throughput constraints.

- A definition of scalable dataflow graphs motivated by the need to meet a fixed throughput constraint for varying problem sizes.

3

- An algorithm classification that makes explicit the impact that data dependencies have in real-time implementations.

- A real-time implementation strategy that decomposes the most problematic algorithms into compositions of more predictable constituents and then uses scalable dataflow graphs and parallel processing to recover timing predictability by mapping data dependent timing uncertainties into the spatial dimension (processors).

Section 2 ends with a discussion of computer architectures, including the single instruction multiple data (SIMD) architecture used in the case studies, and of programming languages, including an argument that functional programming languages, such as SISAL, are better suited than imperative languages for implementing computations described by dataflow graphs on parallel machines.

Two case studies are used to develop the ideas introduced in Section 2. Section 3 describes the implementation of the Modified Gram-Schmidt (MGS) algorithm on a MasPar MP1. The MGS algorithm is commonly employed in antenna-array signal processing to adaptively remove interfering sources. The algorithm produces a factorization of an m by n matrix, called the QR-decomposition, in $O(mn^2)$ operations, and so represents a computationally challenging signal processing algorithm for real-time applications. In Section 3 the MGS algorithm is described mathematically and a dataflow graph is specified. Four parallel mappings to the MP1 are developed and compared among themselves and to a sequential implementation.

Section 4 describes the implementation of the joint probabilistic data association (JPDA) algorithm on a Thinking Machines CM-2. The JPDA algorithm is used in multitarget tracking to associate sensor returns with predicted target tracks. The JPDA algorithm is the computational bottleneck of an object processing sequence whose computational complexity is strongly (exponentially) dependent on the number of tracks present in the data. Section 2 uses this multitarget tracking example to illustrate a potential algorithm partitioning strategy that separates target-return clustering and data association to recover timing predictability for this data-dependent object processing. In Section 4 the JPDA algorithm is described mathematically and a dataflow graph is specified. First, two sequential implementations in C and SISAL are developed and found to yield comparable running times, which is interesting since the SISAL compiler produces C code that is itself compiled. Next a parallel implementation of the JPDA algorithm is developed with the desired scaling properties to recover timing pre-

dictability through the use of processing resources. A final performance comparison is made between the sequential (SISAL) and the parallel implementation of the JPDA algorithm.

Section 5 provides overall conclusions for the MGS and JPDA case studies and discusses future work. The lessons that we learned from the case studies are:

- Communication costs tend to be the limiting factor in obtaining efficient parallel implementations; coarse grain implementations may be forced that violate latency and throughput requirements.

- In SIMD processing, especially when communication costs imply coarse grain implementations, problem replication can be the most efficient parallelization strategy.

- When a single dataflow graph can be mapped to a fixed machine architecture in a variety of ways, the programmer has more flexibility in meeting system memory and timing requirements.

- In SIMD processing, processors often must be turned off and made to stand idle while other processors finish their tasks. This trait limits the efficiency of SIMD processing for some tasks.

- The execution-time uncertainty inherent in object processing can be removed by parallelism.

- Algorithms coded in functional languages often retain more of the inherent parallelism, which can be exploited automatically by a compiler.

- Serial and parallel implementations can be used to provide insights and improvements for each other.

SECTION 2

FUNDAMENTAL CONCEPTS

This paper is concerned with the use of parallel processing in applications that have strict timing requirements. This section defines the fundamental concepts used throughout the paper and provides background material for the case studies of Sections 3 and 4.

ALGORITHMS

An algorithm A is a recipe that describes how to compute an output from a given input. If the output is determined solely by the input, the algorithm implements a function, denoted by F, from the set of possible inputs, denoted by I, to the set of outputs, denoted by O.

There are many ways to present algorithms. In this paper, we have adopted one common practice of presenting an algorithm as a set of directed numerical equations along with pseudocode for loops and other flow control. The equations are directed in the sense that the expression on the right-hand-side of an equation determines the value of the variable on its left-hand-side. A variable may occur on the left-hand-side of no more than one equation. The input variables are the set of variables which do not occur on the left-hand-side of any equation. We designate a subset of the variables as output variables. We also specify the input set I and the output set O.

For a parallel implementation of an algorithm, we are concerned with how long it takes the network to *communicate* the inputs and the outputs, and how long it takes a processor to *compute* the output from the input. Complexity functions $c_I : I \rightarrow \mathbf{R}$ and $c_O : O \rightarrow \mathbf{R}$ are defined that correspond to the amount of data in each input and output ($\mathbf{R}$ denotes the real numbers). For $i \in I$, we may sometimes informally refer to $c_I(i)$ as the *size* or *length* of i. Similar terminology is used for the outputs in O. The operation complexity $c_A : I \rightarrow \mathbf{R}$ gives for an input $i \in I$ the number of operations required to compute the output $o = F(i)$ using algorithm A. Often we just count additions and multiplications for the number of operations. A key issue for real-time implementations, which we will return to later in this section, is whether c_A depends on the actual value of the input $i \in I$

7

or just on the size of i. Similarly, since $o = F(i)$, does the size of o depend on the actual value of the input $i \in I$ or just on the size of i?

An example will illustrate these notions. For an $n = 2^m$-point fast Fourier transform algorithm A, I and O correspond to the set of n-point complex vectors, and F is the discrete Fourier transform. The complexity functions c_I and c_O are taken to be the vector length, i.e., $c_I(\mathbf{i}) = c_O(\mathbf{o}) = n$. Notice in this case, these complexity functions are independent of the input vector $\mathbf{i}$ and output vector $\mathbf{o}$ elements. The complexity function c_A is taken to be the number of real additions and real multiplications in the fast Fourier transform algorithm chosen, which for one version of the algorithm is $c_A = 5nm$.

LATENCY AND THROUGHPUT

In many computing applications, the objective is to minimize the amount of time required to produce a solution. For a problem instance P, if the inputs become available to the processor at time t_i, and the solution is completed at time t_o, then the problem *latency* is given by $l = t_o - t_i$.

Often in real-time applications like sensor processing, a stream of problem instances $P_1, P_2, \ldots, P_i, \ldots$ must be processed. If there is a constant time period p between each problem instance, then the problem *throughput* is given by $1/p$. The problem throughput can be defined when the period between problem instances is not constant using limits, but we do not do so here.

It is customary to express problem throughput in terms of the number of operations per second. For example, if a problem instance involves an algorithm that requires f floating point operations, then the throughput of $1/p$ problems per second implies that the processor must sustain f/p floating point operations per second (FLOPS). The ratio of the sustained throughput on a problem to a peak processor throughput is defined to be the processing *efficiency* for the problem.

REAL-TIME PARALLEL PROCESSING

In a real-time implementation, the application's latency and throughput requirements are specifications that the computing system must satisfy. In the case of a stream of problem instances with period p, if the problem latency requirement $l \leq p$, then the computing system can only be working on one problem instance

8

at a time, and the latency is the governing requirement. Depending on the algorithm, parallel processing can be used to meet strict latency requirements by *splitting* the computation involved in a single problem instance among multiple processors. The splitting can be done along different lines, e.g., the *data parallel* approach partitions the input data among different processors that implement the same function.

If $l > p$, then the computing system can be working on more than one problem instance at a time, and the throughput is the governing requirement. If a given computer can satisfy the latency requirement, then *replication* is a conceptually simple parallel processing approach for meeting *any* throughput requirement. In the replication approach the individual computer is replicated at least l/p times with each copy receiving a successive problem instance. When the first computer finishes the first problem instance, it becomes available to process the next available problem instance, and so on.

Although the replication approach is a conceptually simple solution for high throughput situations, it may not be a practical alternative depending on how much data must be transmitted to load each problem instance on a single computer. Also as the problem size increases, the requirement to move larger amounts of data means that the replication approach often does not scale well, a notion to be made more precise later.

Pipelining is another parallel processing approach that has been used in high throughput applications. In the pipelining approach the computation is broken down into stages with each stage implemented on a separate processor. Data flows from stage to stage and the localization of high speed data is not required.

Every stage must complete its processing in the problem period p and send the results to successive stages. Stages that accept input data are then available to begin processing the next problem instance. Solutions are produced every period by the output stages. Extra communication buffering can be included between processing stages to improve the processing efficiency at the expense of increased problem latency—as long as the application's latency requirement is not violated, of course.

9

DATAFLOW GRAPHS

In high throughput applications, much attention must be paid to the flow of data within the processing system. A natural formalism for reasoning about this flow is the *dataflow graph* of an algorithm. A dataflow graph consists of a set of nodes that corresponds to tasks in the algorithm and a set of directed edges that correspond to the data dependencies between tasks. There are two special kinds of nodes: *source* nodes that provide the inputs from the external world and *sink* nodes that collect external outputs.

Except for the sources and sinks, each node u itself executes an algorithm A_u, where A_u implements a function F_u, that maps inputs from I_u to outputs in O_u. The dataflow principle is that the node can perform its task whenever the required input data is available on all of its incoming edges.

In a dataflow graph, the *granularity* of a node u and input $i \in I_u$ is defined as the ratio

$$\frac{c_{A_u}(i)}{c_{I_u}(i) + c_{O_u}(F_u(i))}$$

of the number of operations $c_{A_u}(i)$ performed at the node for a single computation of A_u to the total amount $c_{I_u}(i) + c_{O_u}(F_u(i))$ of required input and output data. A common word size, e.g., single precision floating point, is used in the granularity expression. Informally, we say a dataflow graph is *fine grain* if the granularity of its nodes is close to 1. Similarly we say a dataflow graph is *coarse grain* if the granularity of its nodes is much greater than 1. Dataflow granularity is important when algorithms are mapped to parallel computers.

SCALABLE DATAFLOW GRAPHS

A parameterized algorithm can be represented by a family of dataflow graphs, one for each value of the parameter. This representation of an algorithm will be called a *parameterized dataflow graph*, and we will characterize notions of complexity by comparing graphs within a family. For example in many applications, the computation and communication complexity of the algorithm depends only on the size of the input data. In the case of the fast Fourier transform algorithm of a specified size, one could imagine a dataflow graph which performs only either a single real addition or multiplication at each node, so that the computational complexity at each node is always one. The parameterized dataflow graph in this representation of the fast Fourier transform algorithm has a dataflow graph for

10

each input size, and the family has the property that the computational complexity of every node in every dataflow graph is always one.

A parameterized dataflow graph is said to be *computation scalable* if one constant bounds the operation complexity of the algorithm executed at each node in every dataflow graph in the family. A parameterized dataflow graph is $f(n)$-*computation scalable* if the operation complexity of the algorithm executed at every node in the dataflow graph specified by parameter n is bounded by $O(f(n))$. There are dual notions of *communication scalable* and $f(n)$-*communication scalable* that bound the amount of data communicated along the edges in a parameterized dataflow graph. A parameterized dataflow graph is *scalable* if it is both computation and communication scalable. If the parameterized dataflow graph is $f(n)$-computation scalable and $g(n)$-communication scalable, and $O(h(n)) = O(f(n) + g(n))$, then the parameterized dataflow graph is $h(n)$-*scalable*.

If the input and/or output degree of a node is a function of the parameter n, as in algorithms that include broadcasting, then ordinarily the parameterized dataflow graph would not be scalable. However, broadcasting can often be replaced by a sequence of "nearest neighbor" communications to obtain a scalable dataflow graph.

Scalable dataflow graphs are of interest because in real-time applications involving problem streams they can be combined with pipelining to maintain a constant problem throughput independent of problem size. The problem latency may or may not increase depending on the structure of the algorithm. The number of nodes in a scalable dataflow graph usually increases as the input size increases. As long as there are processors to accommodate the additional nodes, then the throughput is maintained independent of problem size. In this way scalable dataflow graphs trade space complexity for timing predictability.

MAPPING ALGORITHMS TO ARCHITECTURES

A critical step in the parallel software engineering process is the mapping of the algorithm to the parallel computer. There is a large literature on this subject, most of which is concerned with mappings that minimize the latency of a single problem instance. See for instance [6], which contains 87 relevant references. Other examples include the current research emphasis on developing compilers that implement emerging high-performance computing languages, such as High

Performance FORTRAN [7]. There has been some work on mappings that maximize throughput for pipelined implementations, most notably the work of Bohkari [8].

A *mapping* consists of a *partition* of the nodes in the dataflow graph, often called *clustering*, and an *assignment* of the clusters comprising this partition to distinct processors of the parallel computer. Once the partition is determined, a new dataflow graph can be constructed by coalescing all the nodes in each cluster of the partition into a single node. Original edges between nodes in two different parts of the partition become edges in the coalesced dataflow graph. Multiple edges between coalesced nodes can be combined if desired. A mapping is *consistent* if the communications corresponding to the edges in the coalesced dataflow graph can be supported by the communication network of the parallel computer. Splitting, replication, and pipelining are examples of mapping approaches that were discussed previously.

An integral part of the mapping process is the determination of a schedule that constrains the order in which the various tasks in the original dataflow graph are computed. In a real-time implementation this schedule must have associated with it time limits in which tasks are completed. A key issue is whether the mapping and schedule are static and can be produced at compile time or whether data dependencies (conditionals and data-dependent iteration) force a dynamic mapping and schedule determined at run-time. There can be a significant amount of overhead when mappings and schedules are determined at run-time.

There are two additional issues that bear on the processing efficiency of the resulting implementation. The first is a local consideration especially relevant to pipelined implementations, namely, matching the granularity of the coalesced nodes to the granularity of the processing elements. If a processing element can sustain r operations per second on the coalesced node's task, and its communication links can sustain receiving or sending s words per second, then the *granularity* of the processing element is defined as the ratio r/s. If the granularities are matched, then in the time it takes to complete the processing of the current instance of the coalesced node's tasks, the inputs for the next instance can be received and placed in a buffer and the outputs of the previous instance can be sent to the next processor in the pipeline. The processing element can be kept totally busy by switching between two such buffers.

The second issue is a global consideration having to do with how well matched the "algorithm architecture," represented by the structure of edges in the dataflow graph, is to the architecture of the communication network of the parallel com-

12

puter. In the past, algorithm-specific implementations have been so successful because the machine architecture was tailored to the algorithm architectures. As we consider fixed machine architectures, the choice of algorithm and how it is decomposed into a dataflow graph takes on added importance. The closer these two architectures are the more transparent the mapping can be and the less likely communication bottlenecks will adversely impact the processing efficiency.

DATA DEPENDENCY CLASSIFICATION

To make explicit the impact that data dependencies have in real-time implementations, we develop an algorithm classification based on the types of static and dynamic data dependencies found in target applications. In particular, we identify four data-dependency classes for an algorithm A with inputs I, outputs O, and function F:

1. both c_A and c_O depend only on c_I,

2. c_A depends only on c_I and c_O depends on $i \in I$,

3. c_A depends on $i \in I$ and c_O depends only on c_I, and

4. both c_A and c_O depend on $i \in I$.

A node of a dataflow graph is classified according to the data-dependency classification of the algorithm it implements. The only subtlety is that the input and output complexities are evaluated separately for each of the incoming and outgoing edges of the node.

A class 1 algorithm exhibits the weakest form of data dependency, where only the input size matters and not the actual values of the input, e.g., the fast Fourier transform example given previously in which the processing and communication times are determined once the input length is fixed. For a class 2 algorithm, the actual data values only affect the size of the output and hence the communication time. For example, an algorithm that loops through a list of input numbers and places into the output list only those numbers that exceed a specified threshold exhibits a class 2 data dependency. In a class 3 algorithm, the actual data values only affect the operation count and hence the processing time, as would be the case for an algorithm that searches a data base for an input string and returns a single bit response (yes or no) in any case. Finally, a class 4 algorithm represents

13

the most dynamic case where both the operation count and output size depend on the actual data values.

REAL-TIME SCHEDULING

A dataflow graph that is composed entirely of class 1 nodes and whose source nodes provide inputs of a prespecified size is called a *synchronous dataflow graph*. In synchronous dataflow graphs, the amount of data produced or consumed by every node in the graph and the computation time at every node does not depend on the data. Mappings and schedules of algorithms that can be represented by synchronous dataflow graphs can be determined at compile time. See for instance [9, 10]. Signal processing applications often have algorithms with synchronous dataflow graphs.

Object processing applications usually pose more of a mapping and scheduling challenge because the algorithms involved usually are not represented by synchronous dataflow graphs. Even class 1 algorithms pose challenges for real-time implementations in the case that the complexity-determining input-size parameter is not specified ahead of time. Such situations correspond to a parameterized family of synchronous dataflow graphs, which we referred to previously as a parameterized dataflow graph. The real-time implementation of class 2, 3, and 4 algorithms pose additional challenges. Dataflow graphs that are not synchronous are referred to as *dynamic* dataflow graphs.

There is currently no general treatment of the real-time mapping and scheduling problem for algorithms with dynamic dataflow graphs. Many of these dynamic dataflow graphs have structure that can reduce the amount of scheduling needed at run-time. For example, a dataflow graph may have several large subgraphs that are synchronous and that need no run-time scheduling of nodes within each subgraph. Several techniques have been developed that facilitate this hybrid approach [9, 11]. They reduce the scheduling overhead compared with fully dynamic scheduling; however, they do not result in predictable schedules without additional knowledge about the target application.

Another possible approach is the extension of the real-time scheduling theory [12] to distributed real-time systems. For example, the generalized rate monotonic scheduling (GRMS) theory has been recently extended to distributed systems based on buses and token rings [13]. GRMS theory was developed for the case of multiple tasks sharing a single processor and provides an analytic

framework for guaranteeing timing requirements in a real-time system. As long as system utilization of all tasks lies below the precomputed GRMS bound, then the system will meet its deadlines provided the proper scheduling algorithms are used. In a parallel implementation, the communication resources often are the scarce commodity and they must also be included in the scheduling framework. Challenges arise because of the distributed nature of the communication networks found in the tightly-coupled parallel processors being considered here.

Our approach to the real-time implementation of dynamic dataflow graphs is to use parallel processing to increase timing predictability by mapping the data dependent timing uncertainties into the spatial dimension (processors). This involves identifying in the target application the static and dynamic data dependencies and then developing scalable dataflow graphs for those functions whose computation or communication complexity cannot be specified ahead of time.

In the case of a dynamic processing bottleneck, the approach is to decompose, if possible, the more problematic class 3 and 4 algorithms into compositions of more time-predictable algorithms of class 1 or 2. The processing time of the algorithms in class 1 and 2 are determined only in terms of the input length, and once the input parameters are specified at run-time, then scalable dataflow techniques are used to minimize timing variations. Similarly, in the case of a dynamic communication bottleneck, the approach is to decompose, if possible, the more problematic class 2 and 4 algorithms into compositions of time-predictable algorithms of class 1 or 3, and again appeal to scalable dataflow graphs.

MULTITARGET TRACKING EXAMPLE

For example, in a multitarget tracking application a portion of the processing chain consists of a clustering algorithm followed by the computationally expensive joint probabilistic data association (JPDA) algorithm. Taken together, this is an example of a class 3 algorithm whose running time is strongly dependent on the actual input data.

The clustering algorithm simply partitions the inputs, which consist of existing tracks and sensor returns, into clusters. In total, the output is the same size as the input—just subdivided. Since clustering can be implemented so that the operation count depends only on the input size, the clustering algorithm by itself is an example of a class 1 algorithm. However, the sizes of the various clusters are the most significant data-dependent parameter as far as the follow-on processing

15

is concerned. If each cluster is sent to a separate dataflow node for follow-on processing, then the output size on each edge depends on the value of the input data and the clustering algorithm corresponds to a class 2 dataflow node. For each cluster, the second JPDA phase is also a class 1 algorithm in that the processing and output requirements are determined by the cluster size, and not by any other aspects of the data. Thus the class 3 processing has been decomposed into a composition of a class 2 algorithm and a class 1 algorithm.

For the JPDA algorithm the processing time varies dramatically as the cluster size is increased only incrementally. Thus to reduce the processing bottleneck for a real-time implementation, the JPDA computation could be represented by a scalable dataflow graph parameterized by cluster size and implemented on a parallel processor with a sufficient number of processors to handle some presumed maximum cluster size. Once the cluster size was known at run-time, the proper dataflow graph is selected and the computation completed within the desired latency or throughput requirement independent of cluster size. Our actual implementation of the JPDA algorithm is discussed in detail in Section 4.

COMPUTER ARCHITECTURES

Nearly all uniprocessor computers are based on the von Neumann architecture. Conceptually, there is a central processing unit that reads and writes locations in memory. Both the data and the program are stored in the same memory. The machine repeatedly performs the following actions:

1. fetch the next instruction in the program from memory,

2. fetch the operands specified by the instruction,

3. perform the indicated operation, and

4. write the results to memory.

In this architecture, a program is a time ordered sequence of commands that map one memory configuration into another. A FORTRAN or C program describes such a sequence of commands in a user-friendly syntax.

The Harvard architecture is a variation of the von Neumann architecture for uniprocessors that segregates programs and data by providing a memory only

for instructions and another memory only for data. The Harvard architecture predates the von Neumann architecture and fell from favor because it does not facilitate programs that manipulate programs as data.

One obvious architecture for parallel machines is a generalization of the Harvard architecture. A Single Instruction Multiple Data (SIMD) architecture consists of one instruction memory connected with a number of central processing units each with their own data memory. The operation of each CPU is synchronous in the sense that each CPU is presented with the same instruction at the same time.

The parallel processors used in our case studies have SIMD architectures. The parallel processor used in the Modified Gram-Schmidt studies in Section 3 was a MasPar MP1. The parallel processor used in the Joint Probabilistic Data Association studies in Section 4 is a Thinking Machines CM-2 Connection Machine.

Another important architecture for parallel processors is the Multiple Instruction Multiple Data (MIMD) architecture. As the name suggests, both the instructions as well as the data associated with each CPU can be different in this architecture.

There are two important subclasses of this architecture. In a multiprocessor, a number of von Neumann style CPUs share one memory. In contrast, in a multicomputer, each processor has its own memory that only it can access. Every multicomputer provides a communication network so that each processor can send data to another processor using some kind of message passing protocol.

Our future work will include studies of machines with a MIMD architecture. In particular, we plan to program the Intel Paragon and the Thinking Machines CM-5. The Paragon is a scalable multicomputer system that connects processing nodes in a two-dimensional mesh configuration [14]. The CM-5 is a scalable multicomputer system that connects processing nodes using a so-called fat tree [15]. The Paragon is of special interest, because ARPA has awarded Honeywell, Inc. a contract to package a machine like it for embedded applications such as airborne sensor processing [1].

PROGRAMMING LANGUAGES

Until recently, nearly all high-level computer languages were designed to encode algorithms for machines with a von Neumann architecture. Imperative languages

17

naturally describe computations performed by these machines. This is because programs in imperative languages specify a sequence of commands that transform a memory configuration into another one. C and FORTRAN are examples of the many imperative programming languages designed for sequential machines.

It is also natural to program machines that use a SIMD architecture with an imperative programming language. As with sequential machines, there is one thread of control and a well defined order in which the configuration of the data memory is mapped into a new one. The case studies used two parallel C programming languages: MPL on the MasPar and C* on the Connection Machine.

Because of the large amount of existing programs written in imperative languages for sequential machines, many people have attempted to construct compilers for these languages targeted at machines that have a MIMD architecture. Even after extensive research and development, automatic parallelizing compilers for imperative languages have not met expectations. There is a fundamental reason for this problem: the model of computation on which these languages are based does not map well to a parallel machine that concurrently executes differing instruction streams.

Functional languages provide an alternative to imperative languages particularly suited for parallel processing. Functional languages implement a different model of computation that is based on evaluating mathematical expressions rather than mapping memory configurations to memory configurations. A program is a set of mathematical definitions followed by an expression that is evaluated in the context of the definitions. The value of any expression depends only on the value of its inputs and not on the order in which an expression is defined.

Programs written in imperative languages naturally map only to machines with a single thread of control, but programs written in functional languages naturally map to all kinds of machines. This is because an expression can be evaluated whenever its input values are known, and because the evaluation of every expression is side-effect free. The evaluation of an expression can be scheduled at any time as long as the schedule respects the expression's data dependencies.

There is an intimate relationship between the evaluation of an expression and the notion of computation described by dataflow graphs. In practice, most compilers for functional languages used dataflow graphs as an intermediate representation of programs. Functional languages naturally specify dataflow graphs. This fact makes functional programming languages particularly relevant in our work.

18

Besides the connection with dataflow graphs, there are two other advantages to using functional programming languages. First, algorithms coded in functional languages often retain more of the inherent parallelism, which can be exploited automatically by a compiler. The semantics of functional languages more closely match the underlying mathematics of an algorithm, a fact that will be demonstrated in Section 4. Second, functional programs are determinate in that they compute the same answer no matter how they are implemented, even if the number of processors varies. As a result, one can write and debug an application on any sequential machine, and be confident that it will compute the same answer when run on any parallel machine.[1]

The work reported in this paper has focused on the use of functional programming languages on sequential machines. We chose the language SISAL [16] because there exists a compiler [17] for the language that produces fast code on conventional architectures. Numerical algorithms written in both C and SISAL use about the same amount of CPU time.

---

1 We have slightly overstated our point. Implementations of a functional program on different hardware can produce different answers due to subtleties involving the implementation of floating point arithmetic. However, errors due to this kind of non-determinancy are very rare in functional programs.

SECTION 3

CASE STUDY: MODIFIED GRAM-SCHMIDT

This section describes the implementation of the Modified Gram-Schmidt (MGS) algorithm on a MasPar MP1. The MGS algorithm is commonly employed in antenna-array signal processing to adaptively remove interfering sources. The algorithm produces a factorization of an m by n matrix, called the QR-decomposition, in $O(mn^2)$ operations. We first describe the MGS algorithm mathematically and specify a dataflow graph. We then develop four parallel mappings to the MP1 and compare them among themselves and to a sequential implementation.

These are the lessons for this case study:

- The main lesson is that communication costs tend to be the limiting factor in obtaining efficient parallel implementations, forcing coarse grain implementations that may violate stringent latency and throughput requirements.

- In SIMD processing, especially when communication costs imply coarse grain implementations, problem replication can be the most efficient parallelization strategy. This is less useful when there is a stream of problems, each of which only populates a part of the processing array since filling the processing array increases the latency as compared to an asynchronous MIMD computation.

- When a single dataflow graph can be mapped to a fixed machine architecture in a variety of ways, the programmer has more flexibility to "engineer" the implementation to meet system memory and timing requirements.

- The central trait of SIMD processing—processors executing the same instructions—limits its efficiency. Often processors must be turned off and made to stand idle while other processors finish their tasks.

21

$$
\begin{array}{ll}
\text{Input: } A \in \mathbf{R}^{m \times n} \\
\text{Auxiliary: } A^1, \ldots, A^n \in \mathbf{R}^{m \times n} \\
\text{Output: } Q \in \mathbf{R}^{m \times n} \text{ and } R \in \mathbf{R}^{n \times n} \\
\\
\text{Set } A^1 = A \\
\\
\text{For } j = 1 \text{ to } n \\
\quad \mathbf{q}_j = \mathbf{a}_j^j, \qquad r_{jj} = 1, \\
\quad r_{jk} = 0, & k \in \{1, \ldots, j-1\} \\
\quad r_{jk} = \dfrac{\mathbf{q}_j^T \cdot \mathbf{a}_k^j}{\mathbf{q}_j^T \cdot \mathbf{q}_j}, \qquad \mathbf{a}_j^{j+1} = \mathbf{a}_k^j - r_{jk} \cdot \mathbf{q}_j, & k \in \{j+1, \ldots, n\}
\end{array}
$$

Figure 2. Modified Gram-Schmidt Algorithm

ALGORITHM DESCRIPTION

Let A be an $m \times n$ matrix with linearly independent columns, so necessarily $m \geq n$. The Modified Gram-Schmidt (MGS) method factors A as QR, where Q is an $m \times n$ matrix and R is an $n \times n$ matrix, such that the columns of Q are orthogonal and R is upper triangular with ones on the diagonal. No square-root function is used, so the columns of Q are not necessarily orthonormal. Such a QR pair is unique and called the QR-decomposition of A. The entries in our matrices are real numbers, unless otherwise noted. The MGS algorithm (using complex entries in the matrices) has been used in spatial filtering, for examples, see the references in [18]. For simplicity, we only implemented versions with real entries.

The MGS algorithm is described in [19, Section 5.7.2]. The algorithm is shown in Figure 2 ($\mathbf{a}_j$ denotes the jth column of matrix A). The algorithm computes successive matrices $A = A^1, A^2, \ldots, A^n$, where A^j has the form

$$
A^j = (\mathbf{q}_1, \ldots, \mathbf{q}_{j-1}, \mathbf{a}_j^j, \ldots, \mathbf{a}_n^j).
$$

Each column $\mathbf{a}_j^j, \ldots, \mathbf{a}_n^j$ is orthogonal to the columns $\mathbf{q}_1, \ldots, \mathbf{q}_{j-1}$. At the jth step, $\mathbf{a}_{j+1}^j, \ldots, \mathbf{a}_n^j$ is made orthogonal to $\mathbf{q}_j$. The jth row of R is determined at this step. After the nth step, all of the $\mathbf{q}_j$ are specified and the R matrix is complete.

22

The MGS algorithm computes $\binom{n}{2}$ vector updates, where the vector has length m and the coefficient of the update is the quotient of two dot products. We count additions, multiplications, and divisions as floating point operations. A dot product takes $2m - 1$ floating point operations; the total number of floating operations that are needed is

$$(n - 1)(2m - 1) + \binom{n}{2}(4m). \tag{1}$$

Thus the total computational complexity is $O(mn^2)$.

DATAFLOW ANALYSIS

For each iteration of the loop of the algorithm shown in Figure 2, the value of $\mathbf{q}_j$ is needed for each of the $n - j$ remaining columns, but then these columns are themselves treated independently. We are led to a triangular dataflow graph shown in Figure 3 for $n = 5$. The $\mathbf{q}_j$ produced by the circle nodes must be available to each square node in the jth column. Thus an implementation would either have $\mathbf{q}_j$ broadcasted to each node or passed in a given column from one node to the next. Each circle node sets the value of $\mathbf{q}_j$ and computes the reciprocal of the dot product of $\mathbf{q}_j$ with itself. Each square node computes the dot product of $\mathbf{q}_j$ and $\mathbf{a}_k$ and the update of $\mathbf{a}_k$.

This dataflow graph is computational scalable with respect to n: the amount of computation at a node is independent of n and increasing n by one would add one more column of nodes (with $n - 1$ square nodes and one circle node). The dataflow graph is m-computation scalable: the amount of computation at each node is linear in m (the dot-product calculations). Note that the dataflow graph is m, n-communication scalable: vectors of length m are sent on the edges and the out-degree of the circle node for $j = 1$ is $n - 1$.

PARALLEL IMPLEMENTATIONS

Historically, pipelining is one method that has been used to parallelize the MGS algorithm for real-time processing. For example, MITRE implemented the MGS algorithm for its wide-bandwidth experimental high frequency system (see [18]) using an array of digital signal processing chips with one processing node for each graph node in Figure 3. The architecture is pipelined in both dimensions, with the time available for real-time block processing equal to m times the input
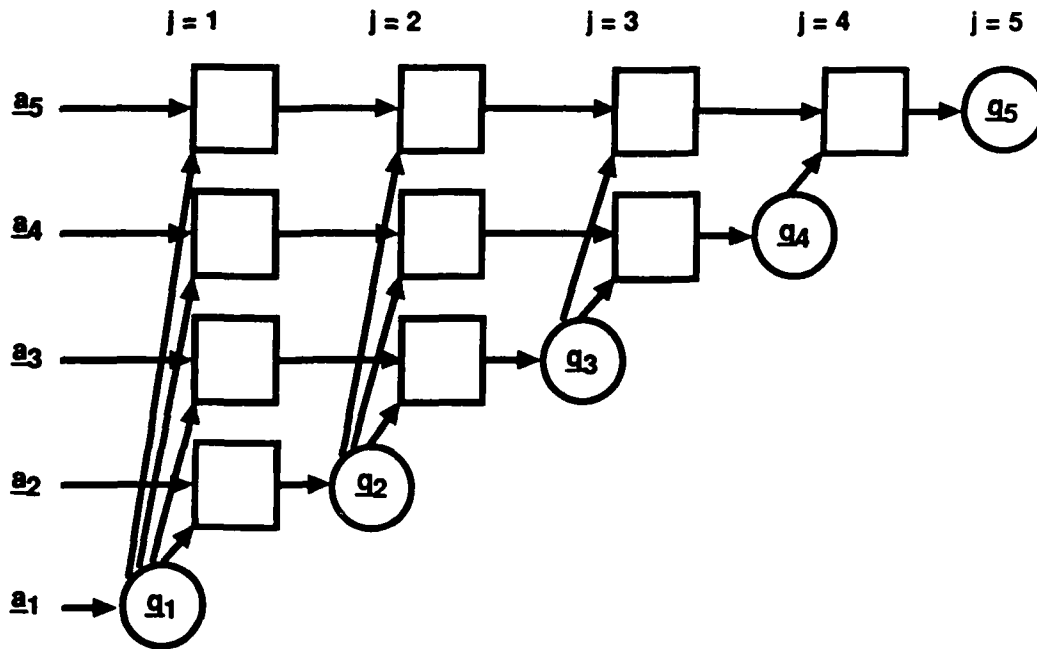
Figure 3. A Dataflow Graph for the MGS Algorithm

sample rate (the system required a 2.048 MHz sample rate with length $m = 128$ complex data vectors). As another example, we implemented the MGS in the functional programming language SISAL, which has a construct called stream that is used to simulate pipelining. However, the current implementation of the SISAL compiler (OSC version V12.9.1) does not correctly implement streams; we plan to investigate this capability of SISAL further when the stream implementation is fixed.

In this section, we present four parallel versions of the MGS algorithm on a MasPar MP1. Three of these versions will parallelize the algorithm using replication; the fourth version will take a pipelining approach. We begin by describing the MP1 and a data parallel version of C called MPL. We then treat each version separately with a description of its encoding and performance. We finish with some comparisons of the four versions.

24

The MP1 Machine

The MP1 consists of a front-end DECstation running ULTRIX and a back-end comprising an Array Control Unit (ACU) and a Data Parallel Unit (DPU). The ACU does some serial computation and broadcasts the instruction sequence to the DPU; it has 128 KBytes of RAM. The DPU has $2^{13} = 8192$ 4-bit processing elements (PEs) in a 64×128 grid. The PEs can be referenced by either their x- and y-coordinates or by a single integer from 0 to 8191. The latter representation treats the DPU as a vector of PEs of length 8192. Each PE has 16 KBytes of RAM. Floating point operations are done by microcode; there are no floating point chips.

There are three methods of communication for the PEs. First, there is a bus that connects the PEs to the ACU. This bus is used to broadcast instructions or data to all the PEs. The second method of communication is the XNet. This network supports communication along the four grid directions (N, S, E, W) and the four diagonal directions (NE, SE, SW, NW). Communication in these directions uses wraparound, so tnat in fact the grid is a torus. The "X" is due to the fact that, even though a PE communicates with its eight nearest neighbors, in actuality there are only four communication paths coming out of the PE in the NE, SE, SW, and NW directions. Thus there is some freedom in which path is used to send data East, for example. The last method of communication is a global router that allows arbitrary communication between PEs. This method is by far the slowest and was not used in any of our implementations.

The MPL Language

We programmed in MPL, which is MasPar's dialect of C. MPL is very close syntactically to C. It adds parallelism by defining `plural` variables and some communication routines on such variables. A key point to remember is that the shape of these variables is the same as the shape of the underlying PE array, that is, either a 64×128 grid or a vector of length 8192.

A variable declared `plural int foo`, for example, results in a 32-bit integer variable `foo` defined for each PE. This variable could be used in arithmetic operations exactly as scalar variables. The global communication syntax is very straightforward. For example, `+= foo` would result in all of the (currently active—see below) values of `foo` to be added together and placed in the ACU. This global reduction uses the ACU-PE bus.

Since every PE executes the same instruction, one way to control processing is to make PEs active and inactive. For example, in the conditional statement

$$\texttt{if (foo == 1)}$$

the currently-active PEs would be further divided into two subsets; which subset a PE is in depends on whether its value of **foo** is 1 or not. Those PEs whose value is 1 would execute the body of the conditional, while the other PEs would execute the body of the **else** clause. Thus, if there is no **else** clause, this second set of processors would be inactive.

Broadcast communication is done via the **proc** construct: **foo = proc[23].foo** would set each currently-active PE's value of **foo** to the value of **foo** in PE number 23. Xnet communication is denoted similarly. For example, **xnetS[5].foo** refers to the values of **foo** that are five PEs in the South direction. Depending on how we use this construct, we could be sending or retrieving data. There are similar Xnet functions for all eight directions. Furthermore, there are constructs that perform copying or pipelining. In copying, the values are copied into all intermediate PEs. In pipelining, the intermediate PEs are ignored (and must be inactive). More general communication using the **router** function was not used for our implementations.

As a comparison of timing for the three different communication constructs, we used the values supplied by MasPar in the MPL User Guide [20]. In all cases, we concentrate on sending a 32-bit word. The timings are given in clock cycles, where for our machine the cycle is 80 nanoseconds. The **proc** command takes 36 clock cycles. The Xnet timings depend on whether copying or pipelining was used and whether the direction is one of N, S, E, W, or one of NE, NW, SE, SW. For example, copying north using the **xnetcN** takes $75 + k$ clock cycles, where k is the distance traversed. The same command in the NE direction takes $175 + 5k$ cycles. The **router** command takes about 5000 clock cycles, which is why its use was avoided in our implementations.

Four MPL Versions

The dataflow graph in Figure 3 can lead to several different parallel implementations. As some examples, we could map each node to its own processor, or each row to its own processor, or even each element of the m-vectors to their own processors. This gives the programmer some flexibility in meeting system timimg and memory requirements. In the past, a special purpose architecture would have been based on the dataflow graph. In the current context of using commercial

26

processors, we must rely on the flexibility that the algorithm affords us to make efficient use of fixed processing resources.

We in fact investigate four different versions of the MGS algorithm in MPL. All of these examples use the data parallel view of computation consistent with the MP1 as a SIMD machine:

1. Map the $m \times n$ matrix onto an $n \times m$ grid of PEs. We replicate to do $64/n$ problems.

2. Map the $m \times n$ matrix onto m consecutive PEs. We replicate to do $128/m$ problems.

3. Map the $m \times n$ matrix onto single PE. We replicate to do 8192 problems.

4. Map the $m \times n$ matrix onto m consecutive PEs, but use n separate rows of PEs to simulate pipelining.

We compare these four versions at the end.

Version 1: $n \times m$ Replication

Our first version of the MGS algorithm implemented in MPL maps the $m \times n$ matrix A onto an $n \times m$ grid of PEs (we transpose to take advantage of the longer dimension of PEs, since $m \geq n$). Thus we are limited to $n \leq 64$ and $m \leq 128$. This mapping allows problems to be replicated: up to $64/n$ problems can be solved simultaneously. The mapping and replication is shown in Figure 4. The shaded portion of the grid represents unused processors. (We could also replicate horizontally and solve a factor of $128/m$ more problems, but this was not tested.)

The code that computes the MGS in MPL is shown in Figure 5. There is a single loop of size n; each iteration corresponds to a column of nodes in Figure 3. During the jth iteration, the jth column of A^j (the **plural** variable A) is passed to the remaining columns. Each of these columns then computes both the dot product with the jth column and the dot product of the jth column with itself. The dot product is computed by doing a component-wise multiplication in parallel and then doing a scan that accumulates the sum along the column: the final element of the scan computation would contain the dot product (this is the scanAddf function). Finally, A is updated for the next iteration and the appropriate row

27

64 x 128 = 8192 Processors
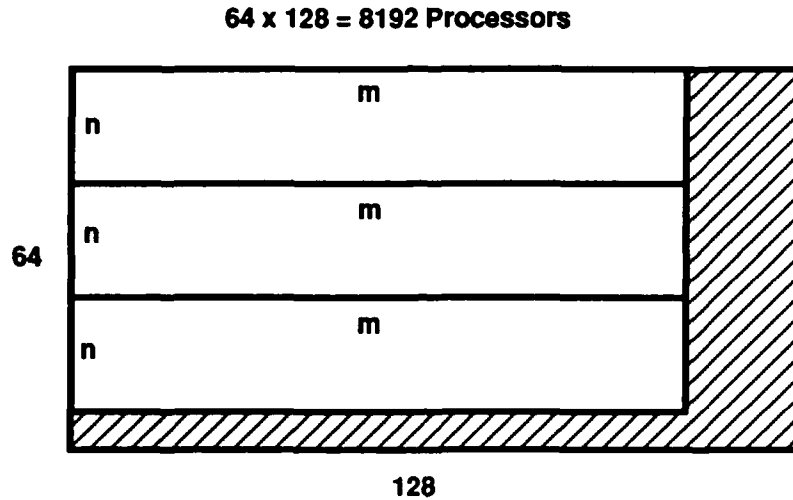
Figure 4. Partitioning the MP1 for $n \times m$ Replication

of the R matrix is filled. The matrix R is stored in an $n \times n$ grid of processors (one matrix entry per processor). As such, we cannot easily use the value of R to update A^j.

Although the code doesn't show it, $64/n$ problems are being solved simultaneously. The `j_per_prob` variable is a `plural` variable that associates the correct column indices with the processor addresses. This variable is easily pre-computed. Thus the conditional using `j_per_prob` makes some of the PEs inactive, because there is no **else** clause.

In Table 1, we show the computation times for running this version for various values of n and m. As we expect, this mapping removes the dependency on m to a large degree. Furthermore, the time dependence on n is linear. Surprisingly, even though we would expect some dependence on n in the `xnetcS[n-1-j]` command (since the distance depends on n), the dependence for just that part of the computation also grew only by a factor of two from $n = 8$ to $n = 16$ rather than the expected factor of four. The reason for this is that, although the number of clock cycles needed grows quadratically in n, the coefficient for the linear part is much larger than that of the quadratic part, and so the linear part dominates for these values of n.

28

```
for (j=0; j<n; j++)
    {
    /* Pass jth column to rest of columns */
    if (j_per_prob == j) xnetcS[n-1-j].jcol = A;
    /* qj dot qj */
    qdot = scanAddf(jcol * jcol,vec_segments);
    if (ixproc == m-1) xnetcW[m-1].qdot = qdot;
    /* qk dot qj */
    qkdot = scanAddf(A * jcol,vec_segments);
    if (ixproc == m-1) xnetcW[m-1].qkdot = qkdot;
    /* Update remaining columns */
    if (j_per_prob > j)
    {
    if (ixproc == j) R = qkdot / qdot;
    A = A - qkdot * jcol / qdot;
    } /* end if */
    } /* end for j */
```

Figure 5. MGS Code Fragment (MPL Version 1)

The times in Table 1 are the latencies for computing the MGS algorithm, assuming all $64/n$ problems occur simultaneously. Throughput can be obtained by taking the reciprocal of these numbers and multiplying by the number of problems being solved concurrently. For example, when $n = 8$ and $m = 128$, we get a throughput of $8/0.01156 = 692.0$ problems per second. We ignore the time needed to read and collect the problems.

Version 2: m Replication

The next version of the MGS algorithm maps the $m \times n$ matrix A onto a vector of PEs of length m. That is, we consider the PEs as a vector of length 8192 and use m of them for a single problem. Thus a whole row of the A matrix would map to a single PE. We are limited to $m \leq 8192$. The limitation on n is a function of the PE memory and not of the mapping. This mapping also allows for replicated problems. In fact, $8192/m$ problems can be solved simultaneously. The mapping and replication are shown in Figure 6. The shaded portion of the grid again represents unused processors.

29

Table 1. Computation Time per Problem for Running MGS on a MasPar MP1 (Version 1)

n	m			
	16	32	64	128
8	0.01142	0.01143	0.01148	0.01156
12	0.01872	0.01874	0.01880	0.01894
16	0.02288	0.02290	0.02301	0.02316

The code that computes this version is shown in Figure 7. There are two loops that depend on n. Each iteration of the outer loop computes the dot product of the jth column of A^j with itself. Although the dot product can be computed in parallel for each of the problems, this value is sent to all of the PEs for a given problem one problem at a time. Though seemingly wasteful, applying the proc construct sequentially to the problems is faster than sending the information other ways. The variable num_probs is set to $8192/m$.

The inner loop is similar to that in version 1: we compute the dot product of the jth column with the current column and send it to the necessary PEs. This requires looping through the problem instances (where also the R matrix is filled). Finally, A^j is updated in parallel. The R matrix is stored in the first n processors of each partition. Thus again we could not easily use it for the updating.

In Table 2, we show the computation times for running this version for various values of n and m. As we would expect, the time dependency is quadratic in n,

8192 Processors

m	m	m	

Figure 6. Partitioning the MP1 for m Replication

30

```
for (j=0; j<n; j++)
    {
    /* qj dot qj */
    qdot = scanAddf(A[j] * A[j],vec_segments);
    for (p = 0; p < num_probs; p++)
        {
        if ( (iproc >= m*p) && (iproc < m-1 + m*p) )
            qdot = proc[m-1 + m*p].qdot;
        } /* end for p */
    for (k=j+1; k<n; k++)
        {
        /* qk dot qj */
        qkdot = scanAddf(A[k] * A[j],vec_segments);
        for (p = 0; p < num_probs; p++)
            if ( (iproc >= m*p) && (iproc < m-1 + m*p) )
                {
                qkdot = proc[m-1 + m*p].qkdot;
                proc[j + m*p].R[k] =
                        proc[j + m*p].qkdot / proc[j + m*p].qdot;
                } /* end if */
        /* Update remaining columns */
        A[k] = A[k] - qkdot * A[j] / qdot;
        } /* end for k */
    } /* end for j */
```

Figure 7. MGS Code Fragment (MPL Version 2)

Table 2. Computation Time per Problem for Running MGS on a MasPar MP1 (Version 2)

		m		
n	16	32	64	128
8	1.80131	0.90824	0.46462	0.24823
12	4.09951	2.06640	1.05612	0.56274
16	7.32944	3.69396	1.88710	1.00420

due to the nested loops. The times decrease with m, rather than stay constant as expected. This is because the communication costs in the algorithm depend on the number of problems, which is $8192/m$. Thus doubling m halves the number of problems, and cuts the time in half.

The latencies for doing $8192/m$ simultaneous problems are the times in Table 2 (again ignoring I/O costs). The throughput can be determined by dividing the number of problems solved by this time. The throughput for $n = 8$ and $m = 128$ is $64/.24923 = 257.8$ problems per second.

Version 3: Complete Replication

In our next MPL version, we do not use a data parallel mapping, but rather we map a whole problem to a single PE. That is, we solve 8192 MGS problems simultaneously. In this way we remove any communication between PEs. The code for this implementation is shown in Figure 8. Notice that this code as written could run on a sequential machine. The parallelism is due to the fact that some of the variables are plural.

The computation times are shown in Table 3 for various values of n and m. The complexity in both n and m is what we ould expect, respectively quadratic and linear. The throughput values can be found by dividing 8192 by the times in Table 3. For example, when $n = 8$ and $m = 128$, the throughput

32

```
for (j=0; j<n; j++)
    {
    /* qj dot qj */
    qdot = 0;
    for (i=0; i<m; i++) qdot += A[i*n + j] * A[i*n + j];
    /* Loop through remaining columns */
    for (k=j+1; k<n; k++)
        {
        /* qk dot qj */
        qkdot = 0;
        for (i=0; i<m; i++) qkdot += A[i*n + k] * A[i*n + j];
        /* Update column */
        R[j*n + k] = qkdot / qdot;
        for (i=0; i<m; i++)
            A[i*n + k] = A[i*n + k] - qkdot * A[i*n + j] / qdot;
        } /* end for k */
    } /* end for j */
```

Figure 8. MGS Code Fragment (MPL Version 3)

is $8192/.57768 = 14180.8$. This is far larger than the throughput in the previous two versions, even though the latency is somewhat greater. Of course, this version has by far the largest I/O cost because of the number of problems that must be read in.

Version 4: $n \times m$ Pipelining

In the last MPL version, we implement a pipelining scheme. An $n \times m$ block of PEs will be used. Each row of this block will correspond to a specific matrix similar to the mapping in version 2. However, once each row of PEs updates the appropriate column of its matrix, it passes the whole matrix down to the next row of PEs. The mapping is shown in Figure 9. The first problem is at the bottom because it is the first problem to finish. This version illustrates a fundamental limitation for SIMD processing: the processors that are not needed when problems are being read in or out must be inactive.

Table 3. Computation Time per Problem for Running MGS on a
MasPar MP1 (Version 3)

| | m | | | |
n	16	32	64	128
8	0.07370	0.14563	0.28966	0.57768
12	0.16638	0.32872	0.65383	1.30410
16	0.29631	0.58537	1.16425	2.32225

The code for this version is shown in Figure 10. There are four sections to the code: starting a problem in the pipe, updating the problems, retrieving a solved problem, and passing the results down. The starting and retrieving are done by broadcast via the proc function. The variables front_input and front_output are scalar arrays on the ACU. The computational step is similar to versions 1 and 2 with one subtlety. The variable j_ind is a plural variable that we precompute: it allows a different loop index for each problem. Thus the for loop is in fact a different loop depending on which row of PEs one looks at. Finally, results are passed down the pipeline using one of the Xnet commands.

In Table 4, we show the total computation time in computing n iterations of the pipe for various values of n and m. After the nth stage, the first problem is completed. We do not include the I/O time in reading in a problem, reading out a problem, or passing the problems down the pipe (the latter time was much smaller than the first two). The dash indicates that we could not complete that problem due to memory limitations. We see that this mapping removes the dependency on m and that the dependence on n is quadratic. To obtain throughput, we would divide n by the times in Table 4. For example, when $n = 8$ and $n = 128$, the throughput is $8/0.07591 = 105.4$ problems per second.

Notice that the throughput is smaller than both versions 1 and 2. Part of the reason is that the first stage of the pipe always computes the update for vector q_1. This accounts for a factor of two, which makes versions 2 and 4 comparable.

34

Problem n

Problem n-1

Flow of the Pipe

Problem 2

Problem 1
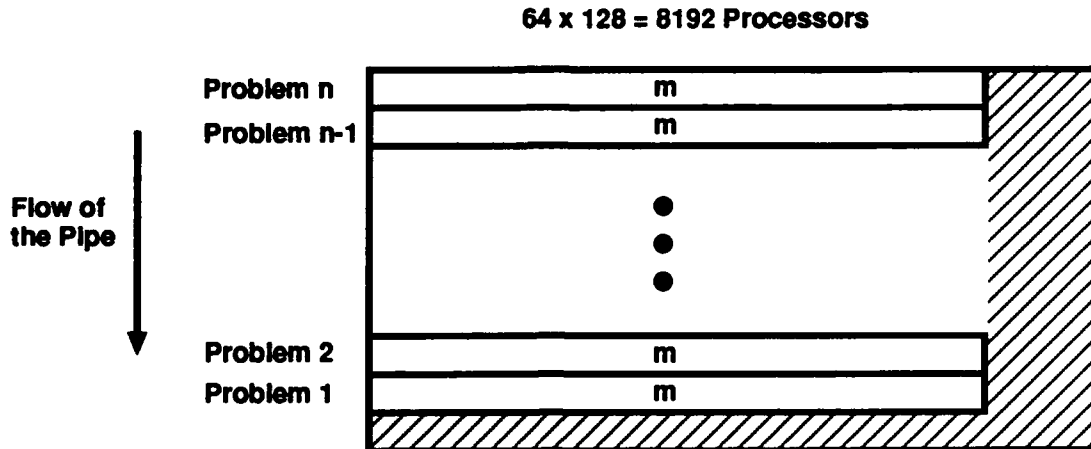
Figure 9. Partitioning the MP1 for Pipeline Implementation

Furthermore, the vector updates are parallelized in version 1, while they are done sequentially in versions 2 and 4.

This version illustrates the SIMD limitation: the processors that are reading in a new problem or reading out a finished problem cannot do so while other processors are computing. In fact, we must turn off any processors that are not executing the current task. The presence of idle PEs degrades the efficiency of the implementation.

PERFORMANCE COMPARISONS

We present two evaluations of the performances of the MPL implementations. The first evaluation compares the latency and throughput of the four versions. The second evaluation compares the performance of two of the versions with a sequential implementation.

Latency and Throughput Comparisons

To compare the four implementations on the MP1, we assume a scenario in which a sequence of problems occur regularly in time. We will assume that $m = 128$ and $n = 8$. In Table 5, we show three measures of performance for each implementation.

35

```
/* Get next problem started */
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        proc[0][i].A[j] = front_input[i*n + j];


/* qj dot qj */
qdot = scanAddf(A[iyproc] * A[iyproc],vec_segments);
if (ixproc == m-1) xnetcW[m-1].qdot = qdot;
for (j_ind = iyproc+1; j_ind < n; j_ind++)
    {
    /* qk dot qj */
    qkdot = scanAddf(A[j_ind] * A[iyproc],vec_segments);
    if (ixproc == m-1) xnetcW[m-1].qkdot = qkdot;
    /* Update remaining columns */
    if (ixproc == j_ind) R[epoch] = qkdot / qdot;
    A[j_ind] = A[j_ind] - qkdot * A[iyproc] / qdot;
    } /* end for plural j_ind */


/* Print out finished problem */
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        front_output[i*n + j] = proc[n-1][i].A[j];


/* Pass results down the pipe */
if (iyproc > 0)
    for (j=0; j<n; j++)
        A[j] = xnetN[1].A[j];
```

Figure 10. MGS Code Fragment (MPL Version 4)

36

Table 4. Computation Time per Problem for Running MGS on a MasPar MP1 (Version 4)

	m			
n	16	32	64	128
8	0.07536	0.07545	0.07559	0.07591
12	0.17020	0.17038	0.17066	0.17143
16	0.30310	0.30342	0.30400	—

The first value is the problem throughput, that is the number of problems per second that the version could support. The reciprocal of this value is the problem period. The second value is the worst-case latency of the problems solved concurrently, that is, the maximum time that one would have to wait for results. These numbers are two times the entries in Tables 1 to 4 for $n = 8$ and $m = 128$. Lastly, we measure the amount of storage needed to store the problems before they are computed. Here we assume that a floating point number takes 4 bytes, so that the space needed for one 128×8 matrix is 4 KBytes.

Since we are ignoring all input and output costs to the machine in this comparison, it is dangerous to draw definitive conclusions. However, one point is clear. The price paid for the highest throughput in version 3 comes with a price: longer latency and very large buffers. Furthermore, we see again a limitation of SIMD processing. All of the problems in version 3 must be executed simultaneously; we could not be reading in new problems as we finish executing other problems.

Comparison of Parallel versus Sequential Implementations

We compare two of the MPL versions—versions 1 and 3—with a sequential version. All of the versions used randomly generated problems with n fixed at 8 and m equal to 16, 32, 64, and 128. The MPL versions used version 3.2.14 of the MPL compiler on a MP1 running Ultrix 4.3. The sequential program was written in C but used the same code as that for version 3 (shown in Figure 8);

Table 5. Comparisons of the Four MP1 Implementations ($n = 8$ and $m = 128$)

Version	Throughput	Latency (sec)	Buffer Size (KBytes)
1	692.0	0.02312	32
2	257.8	0.49846	256
3	14,180.8	1.15536	32768
4	105.4	0.15182	4

its compiler was version 2.1 of the Mips cc compiler. The C version was run on the front-end workstation for the MP1 and was compiled with the -O optimizing flag. In all three implementations, we measured the time for executing the core calculation. We did not record the time needed for I/O (both at the system level and, in the parallel implementations, from the front-end to the back-end). The problem periods are given in Table 6.

In Table 7, we show an estimate for the millions of floating point operations per second (MFLOPS) rate for the three implementations. We use the number of operations given in equation 1. The C version rates are fairly constant, as are those of MPL version 3. MPL version 1 has rates that increase as m increases. This is because the machine is underutilized in our implementation unless $m = 128$ (recall Figure 4).

There is a factor of 8 increase in MFLOPS of version 1 when $m = 128$ over the C version on the front-end. However, we are solving 8 problems simultaneously, so evidently the 128×8 processors used to compute a single problem instance are equal to the processing on the front-end. For MPL version 3, we get an increase of a factor of about 163 over the C version; however, we are solving 8192 problems simultaneously in order to obtain this increase.

We finally show how efficient the three implementations are in Table 8. Efficiency was defined by dividing the MFLOPS rates by the peak MFLOPS for

Table 6. Comparison of Three MGS Implementations ($n = 8$)

Version	m			
	16	32	64	128
MPL V. 1	0.00143	0.00143	0.00143	0.00145
MPL V. 3	0.0000090	0.0000178	0.0000354	0.0000705
C	0.00147	0.00284	0.00566	0.01123

the machine. For the front-end DECworkstation, we used a peak rate of 10.8 MFLOPS. For the MP1, we used a peak rate of 600 MFLOPS. Both rates were obtained from MasPar marketing literature. We see that the efficiency of the MPL version 1 is low, even when $m = 128$. This is because of the communication costs in the algorithm. The MPL version 4 is very efficient at 37%, because no interprocessor communication is needed. However, we are ignoring all I/O costs inherent in loading 8192 problem instances; we would expect such costs to lower efficiency dramatically.

LESSONS

The implementations of the MGS algorithm in this section illustrate how parallelism can be used to compute traditional block signal processing. Our various MPL versions served to illustrate some of the tradeoffs between execution time and communication time and between time and space. We developed the MPL versions in a straightforward way from the dataflow graph. Each of the several mappings suggested by the graph has advantages; the choice would ultimately depend on the real system's requirements. Our implementations tried to address the signal processing framework of receiving a constant flow of problems to be solved.

We unfortunately did not use many software tools in our implementations, though there were several tools that we wished we had. For example, our various

39

Table 7. MFLOPS for Three MGS Implementations ($n = 8$)

Version	m			
	16	32	64	128
MPL V. 1	1.4	2.8	5.6	11.1
MPL V. 3	223.2	226.1	227.6	228.7
C	1.4	1.4	1.4	1.4

partitioning strategies for the different MPL versions were all done by "by hand;" it would be useful to automate this procedure with a graph-based tool. As another example, there was no easy way for us to use the parallel programming environment and debuggers provided with the MP1 in our software development since the machine was remotely accessed. Current policy at MITRE prohibits X-window clients outside MITRE from being displayed internally; a reworking of this policy is ongoing. Needless to say, availability of the programming environment would have been very useful.

The development time for the MPL implementations was fairly fast once the initial learning curve for the MP1 was taken into account. In fact, once the programs were debugged, a large portion of the time was spent in obtaining accurate and meaningful timing data, rather than in optimizing the code. One exception was the attempt to try different communication strategies in MPL version 2. In fact, hindsight would dictate that the best optimization strategy would be to understand the communication constructs more thoroughly: often the actual costs are both counter-intuitive and hidden from the user.

We observed the fundamental limitation of SIMD processing when a stream of problems must be solved. Since the processors must execute the same instruction stream, we could not divide up the processors by function, that is using some for computation and some for communication. Provided that I/O can be achieved, this limitation can be overcome, as demonstrated by the efficiencies in the MPL version 3.

40

Table 8. Percent Efficiency for Three MGS Implementations $(n = 8)$

Version	m			
	16	32	64	128
MPL V. 1	0.2	0.5	0.9	1.9
MPL V. 3	37.2	37.7	37.9	38.1
C	12.7	13.1	13.2	13.3

In the MP1 implementations described in this section, we found a significant drop in efficiency as the granularity was decreased and the need for interprocessor communication increased. This is not an inherent limitation of the SIMD architecture, but rather a consequence of whether a particular implementation can support concurrent communication and computation and whether these two functions can be balanced. Fine-grain systolic SIMD implementations can achieve very high efficiencies if communication is balanced with computation and the two can proceed concurrently. In our MP1 implementations, communication and computation were not performed concurrently.

For a SIMD implementation to be most efficient, the processing array must be completely filled with a single problem or a collection of identical problems. Finer grain mappings will typically have reduced latency, but the most efficient grain size will be determined, as always, by the capabilities of the processor and communication network. The most compatible parallelization strategy for a SIMD architecture, especially when coarse grain processing is indicated, appears to be problem replication. However for problems arriving in a stream, this implies at least a doubling of the latency over a comparable asynchronous implementation since a group of problems must be buffered and then sent into the processing array together—processing on the first problem to arrive does not start until the last problem in group has arrived. Thus SIMD processing will be most efficient in those applications when the individual problem size matches the array size, and especially when the problem decomposes into a large number of repeated

41

calculations, such as matching a given data set with a variety of templates in a target recognition application.

SECTION 4

CASE STUDY: JOINT PROBABILISTIC DATA ASSOCIATION

This section describes the implementation of the joint probabilistic data association (JPDA) algorithm on a Thinking Machines CM-2. The JPDA algorithm is used in multitarget tracking to associate sensor returns with predicted target tracks. The JPDA algorithm is the computational bottleneck of an object processing sequence whose computational complexity is strongly (exponentially) dependent on the number of tracks present in the data. We first describe the JPDA algorithm mathematically and specify a dataflow graph. We then develop two sequential implementations in C and SISAL and compare their performances with each other. We next develop a parallel implementation of the JPDA algorithm with the desired scaling properties to recover timing predictability through the use of processing resources. We finish by comparing the performance between the sequential (SISAL) and the parallel implementation of the JPDA algorithm.

These are the lessons for this case study:

- The main lesson is that the execution-time uncertainty inherent in object processing can be removed by parallelism. That is, space may be utilized in the form of extra processors in order to make the required time for the problem comparable to other parts of the processing chain.

- Algorithms coded in functional languages often retain more of the inherent parallelism, which can be exploited automatically by a compiler. The semantics of functional languages more closely match the underlying mathematics in an algorithm.

- Serial and parallel implementations can be used to provide insights and improvements for each other. Experience with sequential implementations can result in faster development time for the parallel implementation, and analysis of the parallel implementation can result in faster sequential implementations.

43

ALGORITHM DESCRIPTION

Our case study concerns a portion of multitarget tracking. The goal of multitarget tracking is to create and maintain for each target a sequence of predicted states that accurately reflect the true but unknown state of the target. By *state* we mean the array of numerical quantities that specify the target (location, velocity, and so forth). Prediction is based on some subset of all past information of the targets and the returns; which subset depends on the specific tracking strategy. There are several possible strategies, most of which involve making some association of the returns with the targets. A good overall reference is [21].

We focus on a specific association strategy, namely joint probabilistic data association (JPDA). A weighted average of returns is used to update the predicted state. Only returns that are sufficiently close to the given target are considered. That is, a threshold, or *gate*, is put around each predicted state in state-space; returns outside the gate are not used for updating. Furthermore, targets are not treated independently: if two targets share a return, that is, if a return falls in the intersection of their gates, then the two targets are *clustered* together. For a given cluster, JPDA computes all possible hypotheses for which returns came from which targets. The result is a potentially more accurate and robust tracking procedure; the cost is an exponential explosion in computation (exponential in the number of targets in the cluster).

The input for JPDA on a given cluster is an $n \times (m + 1)$ matrix $P = [p_{ij}]$, where n is the number of targets and m is the number of returns. The values p_{i0} are all equal and are based on clutter assumptions. For $j = 1, \ldots, m$, p_{ij} is based on probability models assuming that the jth return came from target i. In particular, if return j is outside of the gate for target i, then p_{ij} is set to zero.

Figure 11 shows three targets that are being tracked and five returns. The three targets are clustered together because of a return in the intersection of the respective gates. (The actual states are shown for illustration purposes and have no effect on the computation.) Based on the figure, the input matrix is thus

$$P = \begin{bmatrix} p_{00} & p_{01} & p_{02} & p_{03} & 0 & 0 \\ p_{10} & 0 & 0 & p_{13} & p_{14} & 0 \\ p_{20} & 0 & 0 & 0 & p_{24} & p_{25} \end{bmatrix} \tag{2}$$

The goal of JPDA is to compute the weights

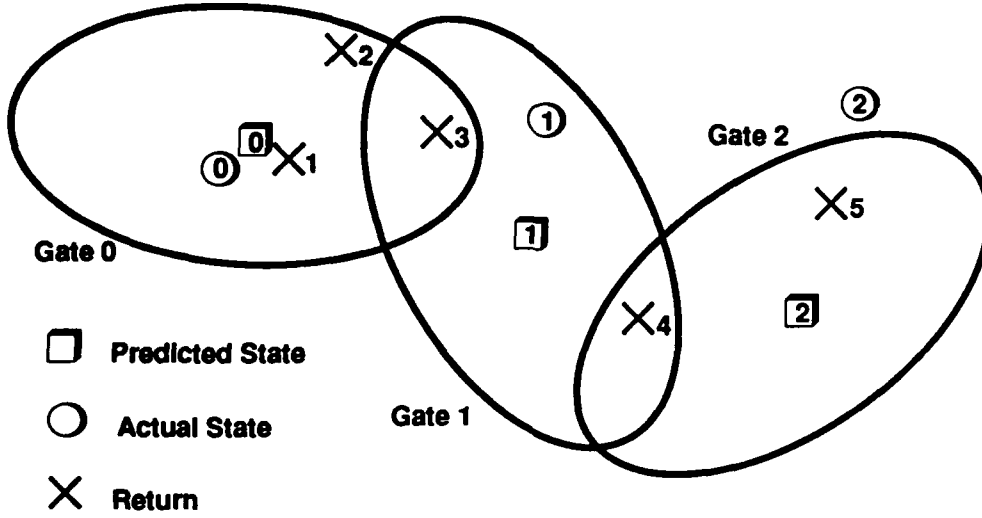$$\beta_{ij} = \text{Prob(target } i \text{ is associated with } j)$$

44

Figure 11. A Cluster for JPDA

and β_{i0}, which is the probability that target i was not associated to any return. The β_{ij} are obtained by a normalization of the α_{ij}, which are defined by

$$\alpha_{ij} = p_{ij} \, \mathrm{per}_0(P_{ij}).$$

The matrix P_{ij} is a submatrix of the matrix P formed by deleting the ith row and the jth column if $j \neq 0$. The function $\mathrm{per}_0()$ is the sum of all products where exactly one element is chosen from each row and at most one element from each column, except for column 0, from which any number can be chosen. This function is related to the permanent of a matrix (see [22]); it is conjectured that calculation of the permanent (and also per_0) is exponentially hard.[2] Note that in fact we must compute $n(m+1)$ permanents.

Recent work at MITRE has applied faster algorithms for computing the permanent of a matrix to the JPDA algorithm (the total time, though, is still exponential); see [23]. We will use the column-recursive JPDA algorithm of [23] in our case study. The algorithm is given in Figure 12 (we use Z_n to denote $\{0, \ldots, n-1\}$). The algorithm computes the α_{ij} by creating an intermediate

---

2 More precisely, calculation of the permanent is #P-complete.

Input: $P = [p_{ij}] \in \mathbf{R}^{n \times (m+1)}$

Auxiliary: $F = [f_{ja}] \in \mathbf{R}^{(m+1) \times (2^n - 1)}$

Output: $B = [\beta_{ij}] \in \mathbf{R}^{n \times (m+1)}$

$$f_{ja}^0 = \begin{cases} \prod_{i \in a} p_{ij} & j = 0 \text{ and } a \subset \mathbf{Z}_n \\ \text{undefined} & 0 < j \le m \end{cases}$$

For $\ell = 1$ to m do

$$f_{ja}^\ell = \begin{cases} f_{ja}^{\ell-1} + \sum_{i \in a} p_{i\ell} \cdot f_{j,a \setminus \{i\}}^{\ell-1} & j < \ell \text{ and } a \subset \mathbf{Z}_n \\ f_{0a}^{\ell-1} & j = \ell \text{ and } a \subset \mathbf{Z}_n \\ \text{undefined} & \ell < j \le m \end{cases}$$

$$\beta_{ij} = \alpha_{ij} / \sum_{i \in \mathbf{Z}_{m+1}} \alpha_{ij} \qquad i \in \mathbf{Z}_n \text{ and } j \in \mathbf{Z}_{m+1}$$

where $\alpha_{ij} = p_{ij} \cdot f_{j, \mathbf{Z}_n \setminus \{i\}}^m$

Figure 12. Column-Recursive JPDA Algorithm

matrix F. This matrix has $m + 1$ rows and $2^n - 1$ columns; the columns are indexed by all proper subsets of the n targets. The zeroth row of F is initialized based on the first column of P. We then iterate on ℓ from 1 up to m. At each iteration, we update the rows of F up to the ℓth row; the ℓth row is just set to the current value of the zeroth row. After the looping, we find the value of $\text{per}_0(P_{ij})$ by looking at the jth row and the subset $\mathbf{Z}_n \setminus \{i\}$.

There are four dimensions that we iterate over in the algorithm. The indices ℓ and j loop through $m + 1$ (though in fact j only loops up to ℓ). The index a runs through all proper subsets of the targets, which total $2^n - 1$. Finally, the index i loops through all possible elements of a, which is at most $n - 1$. Thus the total computational complexity of the algorithm is on the order of $nm^2 2^n$. In what follows, the parallelism that we achieve from this algorithm will come from either parallelizing or unordering one or more of these loops.

DATAFLOW ANALYSIS

Our dataflow analysis is based on the algorithm shown in Figure 12. Notice that for each iteration ℓ, each row of F is updated exactly the same (for those rows being updated). Furthermore, there is no interaction between the different rows of F, except that each row is initialized based on the current value of the zeroth row. Thus, we have in fact $m + 1$ essentially independent computations.

One possible dataflow graph comprises $m + 1$ parallel chains, as shown in Figure 13. The figure is based on the sample cluster shown in Figure 11 ($n = 3$ and $m = 5$). Each column in the figure represents an iteration step ℓ. Each row, or chain, computes one row of F (indexed by j). The shaded boxes represent the undefined values of the F matrix as indicated in Figure 12; the jth row of F is undefined if j is larger than ℓ. Notice though that we could in fact repeat the exact computation in the shaded box that is occurring in the first row. In this way the chains would be uniform in computation and truly independent, provided that we shared or broadcasted the P matrix.

The dataflow graph in Figure 13 is computational scalable with respect to the number of returns m. In particular, increasing m by one results in adding one more chain and increasing every chain in length by one. Also for fixed n, the amount of computation in the existing nodes and all additional nodes remains fixed, independent of the value of m. However, this dataflow graph is not scalable with respect to n; increasing n would produce the same dataflow graph but with exponentially increasing computational requirements. Because a column of the F matrix is passed from node to node, the dataflow graph is communication scalable with respect to m but 2^n-communication scalable with respect to n.

The exponential portion of the algorithm comes from the matrix F, whose rows are length $2^n - 1$ with entries indexed by the subsets of the set Z_n. In particular, at each iteration, f_{ja} is updated by looking at all of the subsets of a that are one less in size. For example, using the P matrix defined in equation 2 and $\ell = 4$, the $\{0, 1\}$ entry of the jth row of F would be updated as

$$f_{j,\{0,1\}} \leftarrow f_{j,\{0,1\}} + p_{14} f_{j,\{0\}} + p_{04} f_{j,\{1\}}$$

(where j must be less than ℓ). This equation shows that not all of the elements of the jth row are needed to update a given entry.

We use this observation to show a finer dataflow graph in Figure 14 for the zeroth chain in Figure 13. Each column of nodes in Figure 14 corresponds to a
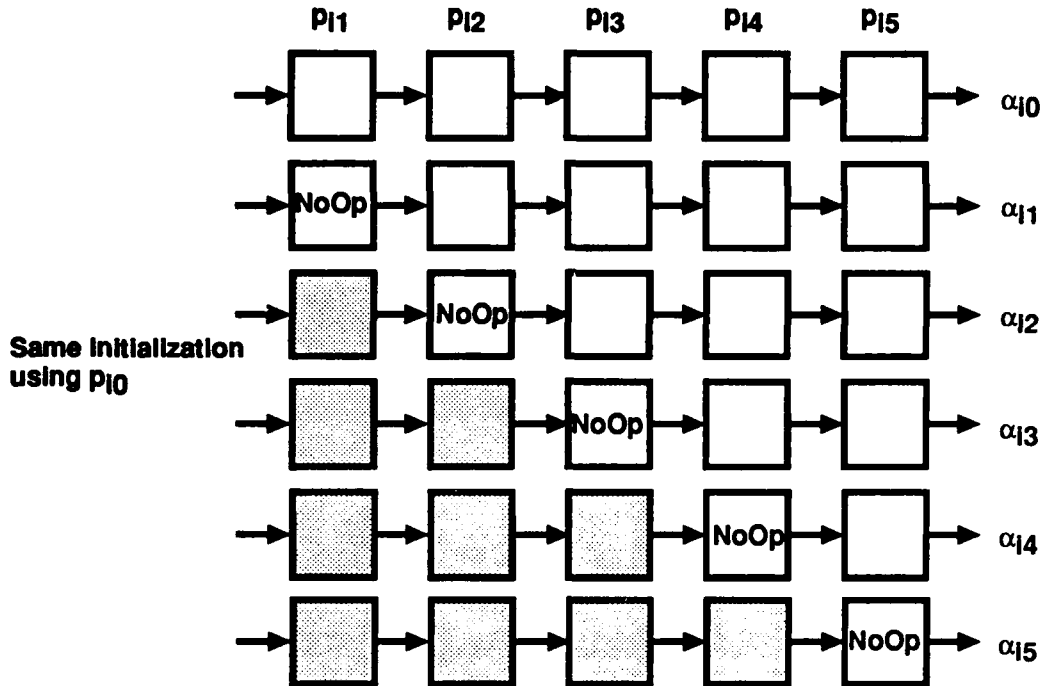
47

Figure 13. A Dataflow Representation for the Column-Recursive JPDA Algorithm

single node of the chain. For clarity, we do not show a horizontal arrow from each node to the next in the same row, although each node indeed depends on its value to the left. The white nodes are shown even though they are not used to compute the answer. Also notice that the corresponding finer dataflow graph for the jth chain of Figure 13 would be shorter if we eliminated the shaded boxes.

The dataflow graph of Figure 14 is scalable in m: increasing m lengthens the chain in a natural way. Increasing n by one doubles the number of nodes. This increase occurs in a very regular manner, namely as a hypercube, which we will exploit in our parallel implementation. However, the dataflow graph is not strictly scalable in n, but rather it is n-scalable. Both the communication and computation at a node increases linearly with n; in the figure this can be seen by the fact that the fan-in at some of the nodes is equal to n.

48

IMPLEMENTATIONS

We present three different implementations of the column-recursive JPDA algorithm. The first was written in the imperative language C which was designed for sequential machines. The second implementation was written in the functional programming language SISAL. This language was designed so that it could be implemented efficiently on both parallel and sequential machines. Our tests of the SISAL version of JPDA were performed on a sequential machine because we did not have access to a parallel machine that runs SISAL programs. The last implementation presented was written in the imperative language C* which was designed for parallel machines. C* is used to program machines produced by Thinking Machines, Inc., which we will use on their CM-2. C* programs contain commands which explicitly state which operations are to be done in parallel.
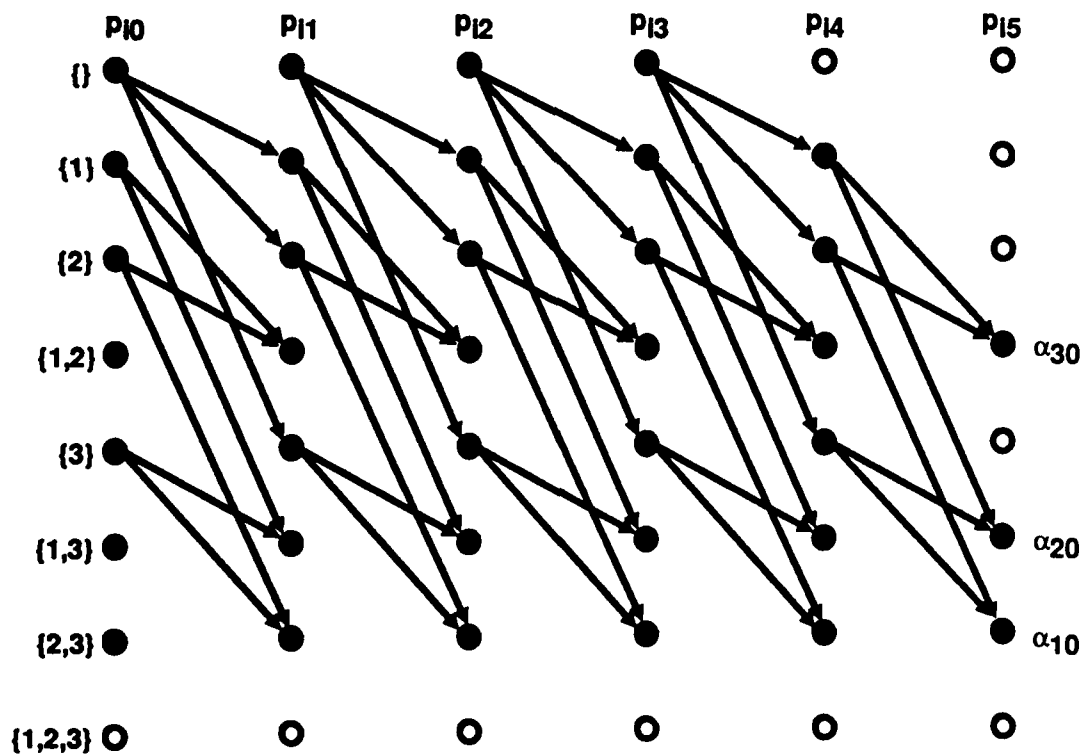
Sequential Implementation: C Version

Figure 14. A Finer Dataflow Graph for the Zeroth Chain in Figure 13

49

$$f_{ja} \leftarrow f_{ja} + \sum_{i \in a} p_{i\ell} \cdot f_{j,a-\{i\}} \qquad a \subset \mathbf{Z}_n$$

```
for (a = two_to_num_rows - 2; a > 0; a--) {
  for (i = 0; ; i++) {
    int k = 1 << i;
    if (k > a) break;
    if (k & a) f[j][a] += p[i][ell] * f[j][a - k];
  }
}
```

Figure 15. JPDA in C

Figure 15 shows a fragment of the C version of JPDA. The fragment contains the innermost two loops of the most time consuming section of the program. This fragment will be compared with both SISAL and C* fragments encoding similar computations later in this paper.

All three versions of JPDA represent sets of integers using an integer. The integer associated with the set a is $\sum_{i \in a} 2^i$. Given the usual binary representation of integers, the membership operation $i \in a$ becomes $2^i \wedge a$, where $\wedge$ is bitwise logical AND. In C, $i \in a$ is implemented by t' ing if ((1 << i) & a) is non-zero, where << is C's shift left operator, and & is ⌐'s bitwise logical AND. The index of the inner for loop in the C fragment is variable i. It ranges over tracks. The inner loop terminates when $2^i > a$.

The program was compiled using the GNU C compiler version 2.4.5 using the -O switch. The CPU time used to execute the entire program for randomly generated input matrices of various sizes is given in Table 9.

Sequential Implementation: SISAL Version

The SISAL version of the JPDA code fragment is given in Figure 16. It consists of two loops, the index of the outer loop ranges over all proper subsets of the set of tracks, and the index of the inner loop ranges over all tracks. It states that a new vector (the jth row of $F^{\ell+1}$) is to be constructed by performing some calculations

which depend on the jth row of F^ℓ. Unlike the C version, it does not specify that the jth row of F^ℓ be replaced with a new value.

The loops displayed are examples of what is called the product form of SISAL's **for** construct. The semantics of this construct imply that the array elements can be computed in any order. A compiler may choose a particular order, or choose to compute some or all of the array elements in parallel. The SISAL version faithfully reflects the fact that the algorithm does not impose a particular order for the computation of array elements.

Let us compare this fragment with the one written in C shown in Figure 15. Due to the sequential semantics of C, the C version of the fragment specifies a particular order in which one must compute and update the array elements. For example, the fact that the index a of the outer loop decreases encodes the fact that its correctness depends on updating f_{ja} before updating every element of row

Table 9. CPU Time in Seconds for Running JPDA on a SPARCstation 10 Model 30 (C version)

Tracks	Returns				
	5	10	15	20	25
10	0.1	0.2	0.4	0.8	1.2
11	0.1	0.4	0.9	1.6	2.5
12	0.3	0.9	2.1	3.5	5.4
13	0.6	2.0	4.3	7.5	11.5
14	1.2	4.3	9.3	16.2	25.0
15	2.7	9.4	20.3	35.4	54.8
16	5.9	20.6	44.5	77.6	119.7

$$f_{ja}^{\ell+1} = f_{ja}^{\ell} + \sum_{i \in a} p_{i\ell} \cdot f_{j,a-\{i\}}^{\ell} \qquad a \subset \mathbf{Z}_n$$

```
function compute_f_matrix_row(f_sub_j: vector; p: matrix;
                             n, two_to_n, ell: integer;
                             returns vector)
  for a in 0, two_to_n - 2
  returns array of
    f_sub_j[a] +
    for i in 0, n - 1
    returns value of sum
      if member(i, a)
      then p[i, ell] * f_sub_j[remove(i, a)]
      else 0.0d0
      end if
    end for
  end for
end function
```

Figure 16. JPDA in SISAL

j that is indexed by a proper subset of a. Reversing the order in which the array elements are accessed would lead to incorrect results.

This is an example of how sequential programs often overspecify a computational process. The correctness of the C program requires a particular order for updating a row of F that is not inherent in the algorithm. A C compiler which generates code for a parallel machine must realize that much of the time-ordering of events given by the program are an artifact of C's sequential semantics. This analysis is very difficult, and the latest generation of compilers do not do a very good job of finding the parallelism, even though more than ten years of research has been directed at this problem.

It is easy for a compiler to discover parallelism in the SISAL version of JPDA because the program does not imply a time-ordering of many computations. However, there is a reason one might be concerned about the efficiency of the

52

SISAL version. The reading of the SISAL code suggests that a new vector be created from f_sub_j. A naive translation of the program would require that memory be allocated each time this vector is created—a very time consuming process. Notice that the algorithm does not refer to the value of f_sub_j once it has been used to create the new vector. The optimizing SISAL compiler has an analysis phase, called *copy elimination* [16], which identifies this pattern of usage and generates code which creates the new vector in the memory previously reserved for f_sub_j. Even with this optimization, however, we found that the size of the run-time image of the SISAL program was roughly twice the size of the C image when applied to the same problem.

As a result of copy elimination and many other optimizations, the SISAL program performs well when compared with the C version even on a sequential machine. Our tests showed that both versions use nearly the same amount of CPU time (the C version was slightly slower, but never by more than 1%). Table 10

Table 10. CPU Time in Seconds for Running JPDA on a SPARCstation 10 Model 30 (SISAL version)

	Returns				
Tracks	5	10	15	20	25
10	0.1	0.2	0.5	0.8	1.2
11	0.1	0.5	1.0	1.6	2.5
12	0.3	0.9	2.0	3.5	5.4
13	0.5	2.0	4.3	7.4	11.5
14	1.2	4.3	9.3	16.2	24.8
15	2.7	9.4	20.1	35.2	54.2
16	5.9	20.7	44.1	77.1	118.7

shows the CPU time for running JPDA on the same workstation used to measure the C run times. All numerical computations are performed using single precision floating point operations. The SISAL programs were compiled with OSC version V12.9.1. It generates C which is compiled into machine language. It invoked Sun's proprietary C compiler version 1.1 with the -O switch.

The SISAL version of JPDA was developed after the C version. The first SISAL version was a direct translation of the C version. It used loop constructs that specify sequential iteration so that the computation was constrained to be performed in the same order as the C version. It ran quite slowly—it used 75% more CPU time. The program was rewritten using the product form of the for loop as shown in 16, and the speedup was dramatic.

We were surprised that the original C version used 40% more CPU time than the fast SISAL version. Upon close examination, we discovered that the original C version had transposed the F matrix. This resulted in more cache misses, and rewriting the program lead to a significant speedup. This is an example of how the functional implementation of JPDA resulted in an improvement in an imperative implementation.

When we compiled the C version using Sun's proprietary C compiler, we found it used 22% more CPU time. As a result, we expected there would be a speedup if the C compiler invoked by the SISAL compiler was changed to the GNU C compiler. Surprisingly, we found that JPDA in SISAL used 8% more CPU time. The authors of the SISAL compiler have tuned their generated C to a particular compiler [24].

Parallel Implementation: C* Version

The previous dataflow analysis suggests several parallelization strategies. For example, we could map the six chains in Figure 13 onto six processors such that no communication between the processors was required (although we would need to store the P matrix at each processor or share it between the processors). In this way, we would achieve a speedup of m. However, because of the exponential computation required at each node, the algorithm will still require exponential time. Also as the number of targets in the cluster fluctuated, the execution time would vary dramatically—a problem for a real-time application that requires predictable running times.

Another approach would be to use a hypercube architecture. A hypercube of dimension n is naturally indexed by subsets of an n-set, which also indexes the columns of F. If we were to map the matrix F onto a hypercube, such that each node received a column of F (length $m + 1$), we would be able to perform the JPDA calculation in polynomial time. This corresponds to mapping each row of nodes in Figure 14 to a single processor; each such row for each of the $m + 1$ chains of Figure 13 would map to the same processor. The price paid is the exponential amount of space that we need in the form of the processors on the hypercube. Such a mapping benefits from the nearest neighbor communication, that is, communication is localized.

We show a mapping with data dependencies for a dimension-three hypercube in Figure 17. Recall that in Figure 13, each parallel chain produced a column of the α_{ij}. In the hypercube mapping, n of the nodes will each produce a row of the α_{ij}; these nodes are the subsets $Z_n \setminus \{i\}$. The white node at the corner is never used by the algorithm (it is the node indexed by Z_n).

Thus, the hypercube architecture achieves a speedup on the order of 2^n at the cost of space 2^n. Such a tradeoff would be useful in practice, since now the time to compute the JPDA algorithm is on the same order as other parts of the signal processing chain—polynomial in n and m—and so easier to manage at the systems level. Furthermore, notice that in the hypercube mapping there are still $m + 1$
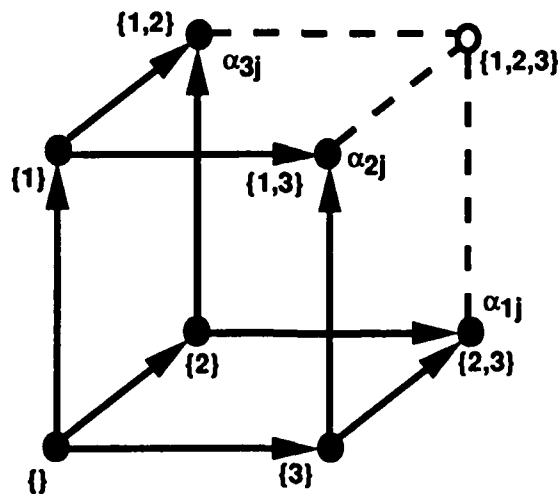

Figure 17. Hypercube Mapping of JPDA Vector Operation

55

essentially independent operations at each node, which correspond to the $m + 1$ chains in Figure 13. That is, we could in fact have $m + 1$ parallel hypercubes, for an additional speedup factor of m.

The parallel implementation of the JPDA algorithm that we will discuss uses a machine with a hypercube architecture. Specifically, we use the CM-2 Connection Machine of Thinking Machines, Inc.

The CM-2 Machine

The CM-2 consists of a front-end Unix workstation and a SIMD back-end comprising $2^{13} = 8192$ one-bit processors connected in a hypercube (other sizes exist; everything that we present will reflect the actual machine that we used). There is a 32-bit floating point chip allocated to every 32 processors. The set of 32 processors, the 32-bit floating point chip, and some communication interfacing is called a sprint node. Thus the machine that we used has 256 sprint nodes. Interprocessor communication is faster if it stays within the boundaries of the sprint nodes. However, we did not seek to optimize our code to take this into consideration.

Because the CM-2 is a SIMD machine, we adopted a data parallel view of computation. That is, each processor received a small piece of the data, which it updated through communication with other processors. We chose the CM-2 because its hypercube structure was well-matched to the dataflow analysis for JPDA that we presented above. Furthermore, the communication required by the dataflow graph is nearest neighbor communication along the hypercube.

Another feature of a SIMD architecture is that of setting contexts. Since each processor receives the same set of instructions, one way to control processing is to turn selective sets of processors off. The set of currently active processors is called the context. In our implementation, on average half of the processors were inactive. In fact, the amount of time that a processor is used is a linear function of the binary weight of its physical address.

The C* Language

We programmed in C*, Thinking Machines' dialect of C. The language adds parallelism through parallel variables and communication routines on them. Parallel variables are defined by first defining their shape. In C*, the shapes can be any multidimensional grid (up to 31 dimensions) with the lengths being a power

of 2. Thus one has the flexibility to define a one-dimensional vector of length 8192 (or larger), or a 128 × 64 grid, etc. Although the total size of the shape may exceed the size of the hypercube, it cannot be smaller. (A user desiring to use a smaller piece of the machine must artificially define another dimension and turn off or ignore those positions.) In our example, we defined a shape to be a hypercube of dimension n, the number of targets. In other words, our shape was an n-dimensional grid with length 2 in each dimension (so communication along the grid is equivalent to hypercube communication).

A parallel variable is defined in terms of its shape, e.g., int:hypercube would mean a 32-bit integer variable with the shape hypercube. Operations on parallel variables are the same as scalars (adding, multiplying, etc.) with the same syntax. An important point is that only variables of the same shape may usually be combined.

There are several ways to communicate between processors. For example, there are send and get commands that transmit information between arbitrary processors. There are also grid communication calls, which respect the actual grid (as defined by the shape). These can be with or without wraparound. Needless to say, there are also many other features, such as scan, spread, and so forth, which we did not need for our implementation.

If the shape is larger than the actual machine size, then virtual processors (VPs) are invoked. Their usage is transparent to the user and allows for easy scalability. For example, when we set $n = 13$, our matrix F maps exactly onto the processors: each processor would receive a length $m + 1$ column vector of F. If $n = 14$, then each processor receives two columns. For $n = 15$, each processor receives 4 columns, and so forth. The ratio of the total size needed to the size of the machine is called the VP ratio. The advantage to a high VP ratio is that potentially a larger fraction of the computation is being done within the sprint nodes, and so interprocessor communication cost is reduced.

C* Implementation

Figure 18 shows one way to encode the code fragment in C* that we used previously to demonstrate C and SISAL. Our variable F is defined as an array of float:hypercube. The matrix P is defined on the front-end, that is, it is a serial two-dimensional array. The variable dummy is used to preserve the current value of F until it can be updated. The where statement turns off all processors— subsets—that do not contain target i. The other half of the processors look in

57

$$f_{ja} \leftarrow f_{ja} + \sum_{i \in a} p_{i\ell} \cdot f_{j,a-\{i\}} \qquad a \subset \mathbf{Z}_n$$

```
dummy = f[j];
for (i=0; i<num_rows; i++)
  where (pcoord(i) == 1)
    dummy = dummy + (float:hypercube)p[i][ell] *
              from_grid_dim(&f[j],(float:hypercube)0.0,i,-1);
f[j] = dummy;
```

Figure 18. JPDA in C*

dimension i at their nearest neighbor and use the value of f[j] located there. Thus every subset sends its value to all subsets that are one element larger, as the dataflow graph in Figures 14 and 17 indicate.

We show some timing results for our C* implementation of JPDA in Table 11. We timed the total elapsed time for the subroutine call that computes JPDA. The time that just the back-end was busy is not shown but was comparable; in other words, almost all of the time in the routine is used by the back-end. Several items are immediately apparent. Since the physical hypercube of the machine has dimension 13, we expect that the complexity for $n \leq 13$ should be on the order nm^2. Indeed this appears to be the case. Once $n > 13$, we see for fixed m the doubling of times that would be consistent with the larger VP ratios. However, the times are slightly less than what would be expected, probably due to the increased usage of the sprint nodes.

Our results used 32-bit floating point operations. We tried 64-bit floating point operations, but the times were about a factor of ten slower. The reason is that the floating point units on our CM-2 were computing serially the longer precision floating point operations. Versions of the CM-2 with 64-bit floating point chips do not have this feature.

We tried to isolate how much time was spent communicating between processors in our implementation and how much time was spent actually computing.

58

Table 11. Elapsed Time in Seconds for Running JPDA on a Thinking Machines CM-2 (C* version)

Tracks	Returns				
	5	10	15	20	25
10	0.11	0.34	0.67	1.14	1.73
11	0.13	0.37	0.77	1.26	1.93
12	0.14	0.39	0.81	1.34	2.05
13	0.15	0.43	0.88	1.47	2.24
14	0.29	0.81	1.65	2.78	4.25
15	0.57	1.59	3.26	5.61	8.49
16	1.16	3.23	6.62	11.25	17.20
17	2.35	6.58	13.44	22.94	35.08
18	4.86	13.53	27.63	47.17	72.05

Our informal check indicated that roughly two thirds of the time was spent on communication, namely the from_grid_dim call.

PERFORMANCE COMPARISONS

We produced two different implementations of JPDA, one implemented in the C* programming language and the other in SISAL. We compare the two implementations of JPDA using randomly generated problems with 20 returns and a varying number of targets. The results are summarized in T ' le 12. The C* program was compiled using version 6.0.3 (196) of the C* compiler on a Sun -/280 running SunOS 4.1.3. The SISAL program was run on the front-end of the CM-2 and was compiled with OSC version V12.9.1. The timing procedure was slightly different. The SISAL program times are total user time for the run calculated exterral to the program; the C* times are total elapsed time calculated within the program.

In Figure 19, we show an estimate for the millions of floating point operations per second (MFLOPS) rate for the two implementations: the C* version on the CM-2 (black) and the SISAL version on the same Sun-4 (gray) (we used the same executable file as that used on the Sparc10 for the results in Table 12). The purpose of the figure is not so much to compare the actual computation rates on the platforms, but to indicate how this rate changes with the problem size. Here we fixed the number of returns to be 20 and varied the number of targets. We calculated the number of operations based on $(m^2 + m + 1)n2^{n-1}$.

Table 12. Comparison of Two JPDA Implementations

Version	Number of Targets								
	10	11	12	13	14	15	16	1⁻	18
SISAL	1.8	3.8	7.9	16.8	36.0	76.8	172.3	377.2	826.2
C*	1.4	1.6	1.5	1.7	3.0	5.8	1ⁱ 5	23.2	47.4

When n is less than 13, the C* version is underutilizing the machine, since the desired hypercube is less in size than the actual machine. In order to reflect the fact that we could then be solving more than one problem at a time, the dashed line shows that expected number of MFLOPS for the C* version for the smaller problem sizes.

Notice how in the SISAL version and in the C* version up to $n = 13$, the number of MFLOPS stays roughly constant. This indicates that the efficiency of these implementations does not change with the problem size. We show efficiency in Figure 20. Efficiency is defined as MFLOPS divided by the peak MFLOPS for the machine. For the Sun-4 we used a peak rate of 8.335 MFLOPS (obtained from a Sun representative). For the CM-2 we used 896 MFLOPS for $n \geq 13$ and this value divided by 2, 4, and 8 for $n = 12, 11, 10$, respectively. The CM-2 value is based on a 3.5 MFLOPS rate per each 256 Sprint nodes (extrapolated from Connection Machine literature).
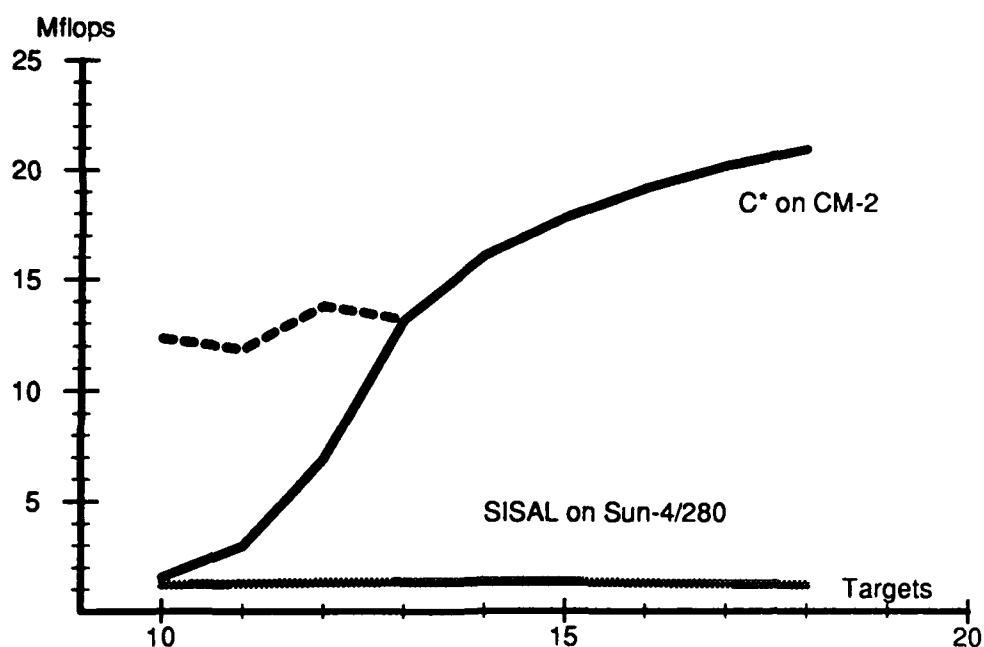
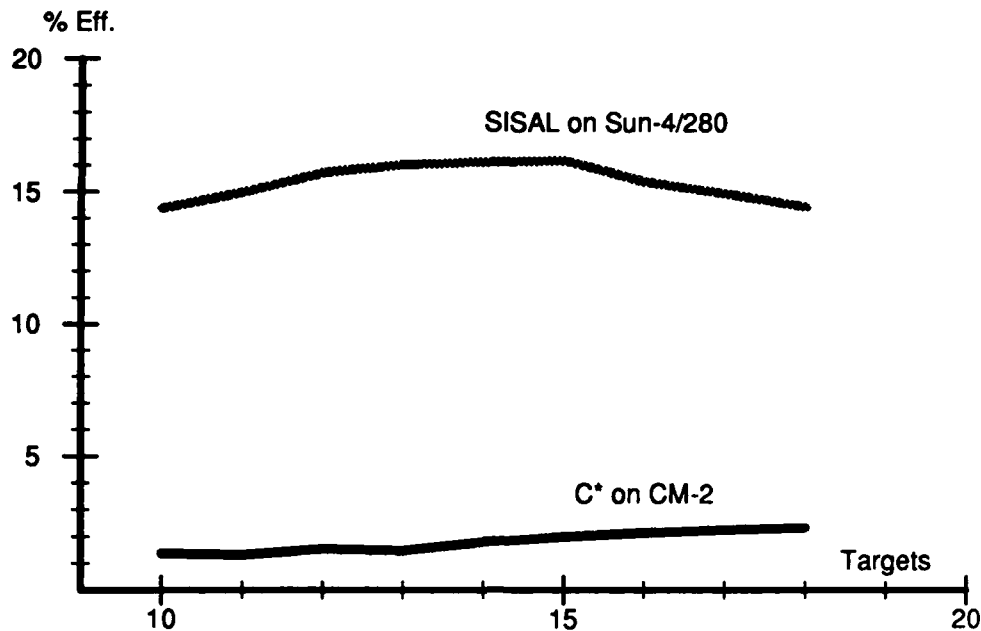Figure 19. MFLOPS for Two JPDA Implementations

Figure 20. Efficiency for Two JPDA Implementations

The efficiency is higher in the sequential version, due to the lack of communication costs. Once $n > 13$, we see that the efficiency of the C* version increases. This rise is due to the amount of communication that becomes localized in the sprint nodes. Memory limitations prevented us from testing how far this growth in performance extends.

LESSONS

The various implementations of the JPDA algorithm illustrate several additional lessons. Primarily, this algorithm demonstrates dramatically the space-time tradeoff, and in particular, how adding parallelism can reduce the execution time to manageable (polynomial) levels. The required parallelism was only apparent on viewing a fine enough dataflow graph—one which was scalable in the problem dimension that caused the exponential time complexity.

Another theme of these implementations is how much the sequential and parallel paradigms interact. For example, an extensive knowledge of past sequential implementations was useful in both the dataflow analysis and the C* implementations. Indeed, the algorithm was first specified by such an implementation. On the other hand, both the dataflow analysis and the performance analysis for the SISAL version led to some easy improvements in the C version.

We found that the way to write an efficient SISAL program is to start with a parallel algorithm and not with a sequential program. The compiler delivers its best code when a program retains as much of the parallelism inherent in the algorithm as possible.

We found the availability of useful tools wanting. There was only one tool that we were able to extensively exercise. The Parallel Assessment Window System (PAWS) [25] is a program designed to allow comparisons among different machines running a single application. It contains four tools: the application tool, the architecture characterization tool, the performance assessment tool, and the interactive graphical display tool. The application tool translates Ada programs into a dataflow representation using IF1 [26]. The SISAL compiler performs this translation for us. The display tool allowed us to visualize the structure of a SISAL program's dataflow graph. This helped in the understanding of the program. We were unable to make use of the performance assessment tool and the architecture characterization tool; however, it should be noted that we had access only to a preliminary version of PAWS.

There were some tools that we would have liked to have seen. A graph-based visual tool to help show mappings would have been better than the "by-hand" techniques used to do the dataflow analysis. As we mentioned in the modified Gram-Schmidt conclusion, it would have been useful to apply the parallel debuggers that come with the CM-2, but there was no easy way to do this.

With regard to software development time, the existence of the sequential version in C led to fast implementations in both C* and SISAL (all three versions used only 200–400 lines of code). For the SISAL code, a large portion of the time was spent in optimization. On the other hand, for the C* code, the time was spent in refining the analysis portions of the code, i.e., getting better timing data, and so forth.

SECTION 5

CONCLUSION

This report investigated the use of commercial massively parallel computers for real-time sensor processing. We introduced concepts that are relevant to real-time parallel processing including: a definition of scalable dataflow graphs motivated by the need to meet a fixed throughput constraint for varying problem sizes, an algorithm classification that makes explicit the impact that data dependencies have in real-time implementations, and a real-time implementation strategy that decomposes the most problematic algorithms into compositions of more predictable constituents and then uses scalable dataflow graphs and parallel processing to recover timing predictability by mapping data-dependent timing uncertainties into the spatial dimension (processors). The two case studies—the MGS algorithm and the JPDA algorithm—applied these ideas and represented the range from signal processing to object processing. We repeat from the introduction the lessons that we learned:

- Communication costs tend to be the limiting factor in obtaining efficient parallel implementations; coarse grain implementations may be forced that violate latency and throughput requirements.

- In SIMD processing, especially when communication costs imply coarse grain implementations, problem replication can be the most efficient parallelization strategy.

- When a single dataflow graph can be mapped to a fixed machine architecture in a variety of ways, the programmer has more flexibility in meeting system memory and timing requirements.

- In SIMD processing, processors often must be turned off and made to stand idle while other processors finish their tasks. This trait limits the efficiency of SIMD processing for some tasks.

- The execution-time uncertainty inherent in object processing can be removed by parallelism.

65

- Algorithms coded in functional languages often retain more of the inherent parallelism, which can be exploited automatically by a compiler.

- Serial and parallel implementations can be used to provide insights and improvements for each other.

Although substantial insights were gleaned from the two case studies, much work remains to be done. An end-to-end demonstration that incorporates simultaneously both signal processing and object processing is needed and would most likely require MIMD processing, instead of the SIMD processing considered in this paper. Also the actual real-time requirements of a specific application should more directly influence the parallel mapping chosen. In particular, a software engineering process for real-time applications needs to be formalized in which the processing and communication kernels of the algorithm are first benchmarked on the target machine, and then a mapping is designed using this information to meet prescribed latency and throughput constraints. Commercial-grade tools that purport to make this software development process either easier or the resulting software more portable need to be evaluated, and the fall-off in performance over hand-tuned software quantified.

To broaden our treatment of the real-time parallel processing problem, we plan to implement a scaled-down version of a wide-area, high-resolution synthetic aperture radar (SAR) surveillance system on a variety of high performance computers, including MIMD architectures such as the Intel Paragon and the Thinking Machines CM-5. Such a surveillance system incorporates signal processing to form the SAR image and object processing in the form of automatic target recognition. To form the SAR image, we plan to implement the new Planar Subarray Processing (PSAP) algorithm [5]. We will investigate using parallel programming environments such as Parallel Virtual Machine (PVM) for obtaining portable implementations and determine the resulting impact on application performance. A software architecture will be developed that allows the proper clustering of tasks to match the underlying machine granularity as the code is ported between different platforms.

We are particularly interested in a real-time implementation of a version of the PSAP algorithm that integrates automatic target recognition within the image formation signal processing chain. This so-called decision-directed image formation may be required to reduce the implementation complexity of wide-area, high-resolution surveillance systems to practical levels. Such a combination, how-

ever, will pose real-time parallel processing problems of the sort considered in this paper. In particular, combining the signal processing and object processing stages will introduce data dependencies, and hence timing uncertainties, into this integrated processing chain. For this case, we will investigate software techniques for recovering timing predictability, e.g., this paper's approach of trading space complexity for timing predictability or software fault-tolerance techniques that allow a system to recover from software timing faults.

LIST OF REFERENCES

1. Blitzer, F., 1993, "Militarized Touchstone Program," In *1993 IEEE National Aerospace and Electronics Conference*, Volume 1, Dayton, OH: IEEE, pp. 137–143.

2. Lambert, M., editor, 1993, *Jane's All the World's Aircraft, 1993-1994*, Coulsdon, Surrey: Jane's Information Group, Ltd, pp. 452–454, Entry titled: Boeing E-3 Sentry (AWACS).

3. Blake, B., editor, 1993, *Jane's Radar and Electronic Warfare Systems, 1993-1994*, Coulsdon, Surrey: Jane's Information Group, Ltd, pp. 253–254, Entry titled: Joint Surveillance and Target Attack Radar System (Joint STARS).

4. Barile, E. C., R. L. Fante, and J. A. Torres, October 1992, "Some Limitations on the Effectiveness of Airborne Adaptive Radar," *IEEE Transacations on Aerospace and Electronic Systems*, pp. 1015–1031.

5. Perry, R. P., R. C. DiPietro, A. Kozma, and J. J. Vaccaro, April 1994, "SAR Image Formation Processing Using Planar Subarrays," In *Proceedings of the SPIE Conference on Algorithms for Synthetic Aperture Radar Imagery*, Orlando, FL.

6. Chaudhary, V., and J. K. Aggarwal, March 1993, "A Generalized Scheme for Mapping Parallel Algorithms," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 3, pp. 328–346.

7. *High Performance FORTRAN Language Specification*, May 1993, High Performance Fortran Forum.

8. Bokhari, S. H., January 1988, "Partitioning Problems in Parallel, Pipelined, and Distributed Computing," *IEEE Transactions on Computers*, Vol. 37, No. 1, pp. 48–57.

9. Lee, E. A., and D. G. Messerschmitt, September 1987, "Synchronous Data Flow," *Proc. of the IEEE*, Vol. 75, No. 9, pp. 1235–1245.

10. Sih, G. C., and E. A. Lee, June 1993, "Declustering: A New Multiprocessor Scheduling Technique," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 6, pp. 625–637.

11. Ha, S., and E. A. Lee, November 1991, "Compile-Time Scheduling and Assignment of Data-Flow Program Graphs with Data-Dependent Iteration," *IEEE Transactions on Computers*, Vol. 40, No. 11, pp. 1225–1237.

12. van Tilborg, A., and G. Koob, 1991, *Foundations of Real-Time Computing: Scheduling and Resource Management*, Boston: Kluwer Academic Publishers.

13. Sha, L., and S. S. Sathaye, September 1993, "A Systematic Approach to Designing Distributed Real-Time Systems," *Computer*, pp. 68–78.

14. Athas, W. C., and C. L. Seitz, August 1988, "Multicomputer Message-Passing Concurrent Computers," *Computer*, Vol. 21, No. 8.

15. Leiserson, C. E., October 1985, "Fat Trees: Universal Networks for Hardware-Efficient Supercomputing," *IEEE Transaction on Computers*, Vol. 34, No. 10.

16. Cann, D. C., April 1992, "Retire Fortran? A Debate Rekindled," *Communications of the ACM*, Vol. 35, No. 8.

17. _____, April 1992, *The Optimizing SISAL Compiler: Version 12.0*, UCRL-MA-110080, Lawrence Livermore National Laboratories, Livermore, CA.

18. Rorabaugh, T. L., E. K. Pauer, R. A. Games, and D. A. Loeber, 1993, "A DSP Array for real-time Adaptive Sidelobe Cancellation," *Proceedings of International Conference on DSP Applications and Technology*, Cambridge, MA.

19. Dahlquist, G., and A. Bjorck, 1974, *Numerical Methods*, Prentice-Hall. Inc.. Translated by Ned Anderson.

20. MasPar Computer Corporation, 1993, *MasPar Parallel Application Language (MPL) User Guide*, Sunnyvale, California: MasPar Computer Corporation.

21. Bar-Shalom, Y., and T. E. Fortmann, 1988, *Tracking and Data Association*, San Diego, CA: Academic Press.

22. Garey, M. R., and D. S. Johnson, 1979, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, San Francisco, CA: Freeman.

23. O'Neil, S. D., and M. F. Bridgland, to appear, "Fast Algorithms for Joint Probabilistic Data Association," *IEEE Trans. on Aerospace and Electronic Systems.*

24. McGraw, J., 1993, personal communication.

25. Pease, D., A. Ghafoor, I. Admad, D. L. Andrews, K. Foudil-Bey, T. E. Karpinski, M. A. Mikki, and M. Zerrouki, January 1991, "PAWS: A Performance Evaluation Tool for Parallel Programming Systems," *Computer*, Vol. 24, No. 1, pp. 18–29.

26. Skedzielewski, S., and J. Glauert, July 1985, *IF1—an Intermediate Form for Applicative Languages*, Livermore, CA: Lawrence Livermore National Laboratories, Manual M-170.